Kai-Min Chung
Yu Sasaki (Eds.)

LNCS 15489

Advances in Cryptology – ASIACRYPT 2024

30th International Conference on the Theory
and Application of Cryptology and Information Security
Kolkata, India, December 9–13, 2024
Proceedings, Part VI

6 Part VI

INTERNATIONAL ASSOCIATION FOR CRYPTOLOGIC RESEARCH

Springer

MOREMEDIA

Kai-Min Chung · Yu Sasaki
Editors

Advances in Cryptology – ASIACRYPT 2024

30th International Conference on the Theory
and Application of Cryptology and Information Security
Kolkata, India, December 9–13, 2024
Proceedings, Part VI

Springer

Editors
Kai-Min Chung
Academia Sinica
Taipei, Taiwan

Yu Sasaki
NTT Social Informatics Laboratories
Tokyo, Japan

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-981-96-0937-6 ISBN 978-981-96-0938-3 (eBook)
https://doi.org/10.1007/978-981-96-0938-3

Preface

The 30th Annual International Conference on the Theory and Application of Cryptology and Information Security (Asiacrypt 2024) was held in Kolkata, India, on December 9–13, 2024. The conference covered all technical aspects of cryptology and was sponsored by the International Association for Cryptologic Research (IACR).

We received a record 433 paper submissions for Asiacrypt from around the world. The Program Committee (PC) selected 127 papers for publication in the proceedings of the conference. As in the previous year, the Asiacrypt 2024 program had three tracks.

The two program chairs are greatly indebted to the six area chairs for their great contributions throughout the paper selection process. The area chairs were Siyao Guo for Information-Theoretic and Complexity-Theoretic Cryptography, Bo-Yin Yang for Efficient and Secure Implementations, Goichiro Hanaoka for Public-Key Cryptography Algorithms and Protocols, Arpita Patra for Multi-Party Computation and Zero-Knowledge, Prabhanjan Ananth for Public-Key Primitives with Advanced Functionalities, and Tetsu Iwata for Symmetric-Key Cryptography. The area chairs helped suggest candidates to form a strong program committee, foster and moderate discussions together with the PC members assigned as paper discussion leads to form consensus, suggest decisions on submissions in their areas, and nominate outstanding PC members. We are sincerely grateful for the invaluable contributions of the area chairs.

To review and evaluate the submissions, while keeping the load per PC member manageable, we selected the PC members consisting of 105 leading experts from all over the world, in all six topic areas of cryptology, and we also had approximately 468 external reviewers, whose input was critical to the selection of papers. The review process was conducted using double-blind peer review. The conference operated a two-round review system with a rebuttal phase. This year, we continued the interactive rebuttal from Asiacrypt 2023. After the reviews and first-round discussions, PC members and area chairs selected 264 submissions to proceed to the second round. The remaining 169 papers were rejected, including two desk-rejects. Then, the authors were invited to participate in a two-step interactive rebuttal phase, where the authors needed to submit a rebuttal in five days and then interact with the reviewers to address questions and concerns the following week. We believe the interactive form of the rebuttal encouraged discussions between the authors and the reviewers to clarify the concerns and contributions of the submissions and improved the review process. Then, after several weeks of second-round discussions, the committee selected the final 127 papers to appear in these proceedings. This year, we received seven resubmissions from the revise-and-resubmit experiment from Crypto 2024, of which five were accepted. The nine volumes of the conference proceedings contain the revised versions of the 127 papers that were selected. The final revised versions of papers were not reviewed again and the authors are responsible for their contents.

The PC nominated and voted for three papers to receive the Best Paper Awards. The Best Paper Awards went to Mariya Georgieva Belorgey, Sergiu Carpov, Nicolas Gama, Sandra Guasch and Dimitar Jetchev for their paper “Revisiting Key Decomposition Techniques for FHE: Simpler, Faster and More Generic”, Xiaoyang Dong, Yingxin Li, Fukang Liu, Siwei Sun and Gaoli Wang for their paper “The First Practical Collision for 31-Step SHA-256”, and Valerio Cini and Hoeteck Wee for their paper “Unbounded ABE for Circuits from LWE, Revisited”. The authors of those three papers were invited to submit extended versions of their papers to the Journal of Cryptology.

The program of Asiacrypt 2024 also featured the 2024 IACR Distinguished Lecture delivered by Paul Kocher and one invited talk, nominated and voted by the PC. The invited speaker had not yet been determined when this preface was written. Following Eurocrypt 2024, we selected seven PC members for the Distinguished PC Members Awards, nominated by the area chairs and program chairs. The Outstanding PC Members Awards went to Sherman S. M. Chow, Elizabeth Crites, Matthias J. Kannwischer, Mustafa Khairallah, Ruben Niederhagen, Maciej Obremski and Keita Xagawa.

Following Crypto 2024, Asiacrypt 2024 included an artifact evaluation process for the first time. Authors of accepted papers were invited to submit associated artifacts, such as software or datasets, for archiving alongside their papers; 14 artifacts were submitted. Rei Ueno was the Artifact Chair and led an artifact evaluation committee of 10 members listed below. In the interactive review process between authors and reviewers, the goal was not just to evaluate artifacts but also to improve them. Artifacts that passed successfully through the artifact review process were publicly archived by the IACR at https://artifacts.iacr.org/.

Numerous people contributed to the success of Asiacrypt 2024. We would like to thank all the authors, including those whose submissions were not accepted, for submitting their research results to the conference. We are very grateful to the area chairs, PC members, and external reviewers for contributing their knowledge and expertise, and for the tremendous amount of work that was done with reading papers and contributing to the discussions. We are greatly indebted to Bimal Kumar Roy, the General Chairs, for their efforts in organizing the event, to Kevin McCurley and Kay McKelly for their help with the website and review system, and to Jhih-Wei Shih for the assistance with the use of the review system. We thank the Asiacrypt 2024 advisory committee members Bart Preneel, Huaxiong Wang, Bo-Yin Yang, Goichiro Hanaoka, Jian Guo, Ron Steinfeld, and Michel Abdalla for their valuable suggestions. We are also grateful for the helpful advice and organizational material provided to us by Crypto 2024 PC co-chairs Leonid Reyzin and Douglas Stebila, Eurocrypt 2024 PC co-chairs Marc Joye and Gregor Leander, and TCC 2023 chair Hoeteck Wee. We also thank the team at Springer for handling the publication of these conference proceedings.

December 2024

Kai-Min Chung
Yu Sasaki

Organization

General Chair

Bimal Kumar Roy	TCG CREST Kolkata, India

Program Committee Chairs

Kai-Min Chung	Academia Sinica, Taiwan
Yu Sasaki	NTT Social Informatics Laboratories Tokyo, Japan and National Institute of Standards and Technology, USA

Area Chairs

Prabhanjan Ananth	University of California, Santa Barbara, USA
Siyao Guo	NYU Shanghai, China
Goichiro Hanaoka	National Institute of Advanced Industrial Science and Technology, Japan
Tetsu Iwata	Nagoya University, Japan
Arpita Patra	Indian Institute of Science Bangalore, India
Bo-Yin Yang	Academia Sinica, Taiwan

Program Committee

Akshima	NYU Shanghai, China
Bar Alon	Ben-Gurion University, Israel
Elena Andreeva	TU Wien, Austria
Nuttapong Attrapadung	AIST, Japan
Subhadeep Banik	University of Lugano, Switzerland
Zhenzhen Bao	Tsinghua University, China
James Bartusek	University of California, Berkeley, USA
Hanno Becker	Amazon Web Services, UK
Sonia Belaïd	CryptoExperts, France
Ward Beullens	IBM Research, Switzerland
Andrej Bogdanov	University of Ottawa, Canada

Katerina Sotiraki	Yale University, USA
Akshayaram Srinivasan	University of Toronto, Canada
Marc Stöttinger	Hochschule RheinMain, Germany
Akira Takahashi	J.P. Morgan AI Research and AlgoCRYPT CoE, USA
Qiang Tang	University of Sydney, Australia
Aishwarya Thiruvengadam	IIT Madras, India
Emmanuel Thomé	Inria Nancy, France
Junichi Tomida	NTT Social Informatics Laboratories, Japan
Monika Trimoska	Eindhoven University of Technology, Netherlands
Huaxiong Wang	Nanyang Technological University, Singapore
Meiqin Wang	Shandong University, China
Qingju Wang	Telecom Paris, Institut Polytechnique de Paris, France
David Wu	UT Austin, USA
Keita Xagawa	Technology Innovation Institute, United Arab Emirates
Chaoping Xing	Shanghai Jiaotong University, China
Shiyuan Xu	University of Hong Kong, China
Anshu Yadav	IST, Austria
Shota Yamada	AIST, Japan
Yu Yu	Shanghai Jiao Tong University, China
Mark Zhandry	NTT Research, USA
Hong-Sheng Zhou	Virginia Commonwealth University, USA

Additional Reviewers

Hugo Aaronson
Damiano Abram
Hamza Abusalah
Abtin Afshar
Siddharth Agarwal
Navid Alamati
Miguel Ambrona
Parisa Amiri Eliasi
Ravi Anand
Saikrishna Badrinarayanan
Chen Bai
David Balbás
Brieuc Balon
Gustavo Banegas
Laasya Bangalore
Jiawei Bao
Jyotirmoy Basak
Nirupam Basak
Gabrielle Beck
Hugo Beguinet
Amit Behera
Mihir Bellare
Tamar Ben David
Aner Moshe Ben Efraim
Fabrice Benhamouda
Tyler Besselman
Tim Beyne
Rishabh Bhadauria
Divyanshu Bhardwaj
Shivam Bhasin

Amit Singh Bhati
Loïc Bidoux
Alexander Bienstock
Jan Bobolz
Alexandra Boldyreva
Maxime Bombar
Nicolas Bon
Carl Bootland
Jonathan Bootle
Giacomo Borin
Cecilia Boschini
Jean-Philippe Bossuat
Mariana Botelho da Gama
Christina Boura
Pierre Briaud
Jeffrey Burdges
Fabio Campos
Yibo Cao
Pedro Capitão
Ignacio Cascudo
David Cash
Wouter Castryck
Anirban Chakrabarthi
Debasmita Chakraborty
Suvradip Chakraborty
Kanad Chakravarti
Ayantika Chatterjee
Rohit Chatterjee
Jorge Chavez-Saab
Binyi Chen
Bohang Chen
Long Chen
Mingjie Chen
Shiyao Chen
Xue Chen
Yu-Chi Chen
Chen-Mou Cheng
Jiaqi Cheng
Ashish Choudhury
Miranda Christ
Qiaohan Chu
Eldon Chung
Hao Chung
Léo Colisson
Daniel Collins
Jolijn Cottaar
Murilo Coutinho
Eric Crockett
Bibhas Chandra Das
Nayana Das
Pratish Datta
Alex Davidson
Hannah Davis
Leo de Castro
Luca De Feo
Thomas Decru
Giovanni Deligios
Ning Ding
Fangqi Dong
Minxin Du
Qiuyan Du
Jesko Dujmovic
Moumita Dutta
Pranjal Dutta
Duyen
Marius Eggert
Solane El Hirch
Andre Esser
Hülya Evkan
Sebastian Faller
Yanchen Fan
Niklas Fassbender
Hanwen Feng
Xiutao Feng
Dario Fiore
Scott Fluhrer
Danilo Francati
Shiuan Fu
Georg Fuchsbauer
Shang Gao
Rachit Garg
Gayathri Garimella
Pierrick Gaudry
François Gérard
Paul Gerhart
Riddhi Ghosal
Shibam Ghosh
Ashrujit Ghoshal
Shane Gibbons
Valerie Gilchrist

Xinxin Gong
Lorenzo Grassi
Scott Griffy
Chaowen Guan
Aurore Guillevic
Sam Gunn
Felix Günther
Kanav Gupta
Shreyas Gupta
Kamil Doruk Gur
Jincheol Ha
Hossein Hadipour
Tovohery Hajatiana Randrianarisoa
Shai Halevi
Shuai Han
Tobias Handirk
Yonglin Hao
Zihan Hao
Keisuke Hara
Keitaro Hashimoto
Aditya Hegde
Andreas Hellenbrand
Paul Hermouet
Minki Hhan
Hilder Lima
Taiga Hiroka
Ryo Hiromasa
Viet Tung Hoang
Charlotte Hoffmann
Clément Hoffmann
Man Hou Hong
Wei-Chih Hong
Alexander Hoover
Fumitaka Hoshino
Patrick Hough
Yao-Ching Hsieh
Chengcong Hu
David Hu
Kai Hu
Zihan Hu
Hai Huang
Mi-Ying Huang
Yu-Hsuan Huang
Zhicong Huang
Shih-Han Hung
Yuval Ishai
Ryoma Ito
Amit Jana
Ashwin Jha
Xiaoyu Ji
Yanxue Jia
Mingming Jiang
Lin Jiao
Haoxiang Jin
Zhengzhong Jin
Chris Jones
Eliran Kachlon
Giannis Kaklamanis
Chethan Kamath
Soumya Kanti Saha
Sabyasachi Karati
Harish Karthikeyan
Andes Y. L. Kei
Jean Kieffer
Jiseung Kim
Seongkwang Kim
Sebastian Kolby
Sreehari Kollath
Dimitris Kolonelos
Venkata Koppula
Abhiram Kothapalli
Stanislav Kruglik
Anup Kumar Kundu
Péter Kutas
Norman Lahr
Qiqi Lai
Yi-Fu Lai
Abel Laval
Guirec Lebrun
Byeonghak Lee
Changmin Lee
Hyung Tae Lee
Joohee Lee
Keewoo Lee
Yeongmin Lee
Yongwoo Lee
Andrea Lesavourey
Baiyu Li
Jiangtao Li
Jianqiang Li

Junru Li
Liran Li
Minzhang Li
Shun Li
Songsong Li
Weihan Li
Wenzhong Li
Yamin Li
Yanan Li
Yu Li
Yun Li
Zeyong Li
Zhe Li
Chuanwei Lin
Fuchun Lin
Yao-Ting Lin
Yunhao Ling
Eik List
Fengrun Liu
Fukang Liu
Hanlin Liu
Hongqing Liu
Rui Liu
Tianren Liu
Xiang Liu
Xiangyu Liu
Zeyu Liu
Paul Lou
George Lu
Zhenghao Lu
Ting-Gian Lua
You Lyu
Jack P. K. Ma
Yiping Ma
Varun Madathil
Lorenzo Magliocco
Avishek Majumder
Nikolaos Makriyannis
Varun Maram
Chloe Martindale
Elisaweta Masserova
Jake Massimo
Loïc Masure
Takahiro Matsuda
Christian Matt
Subhra Mazumdar
Nikolas Melissaris
Michael Meyer
Ankit Kumar Misra
Anuja Modi
Deep Inder Mohan
Charles Momin
Johannes Mono
Hart Montgomery
Ethan Mook
Thorben Moos
Tomoyuki Morimae
Hiraku Morita
Tomoki Moriya
Aditya Morolia
Christian Mouchet
Nicky Mouha
Tamer Mour
Changrui Mu
Arindam Mukherjee
Pratyay Mukherjee
Anne Müller
Alice Murphy
Shyam Murthy
Kohei Nakagawa
Barak Nehoran
Patrick Neumann
Lucien K. L. Ng
Duy Nguyen
Ky Nguyen
Olga Nissenbaum
Anca Nitulescu
Julian Nowakowski
Frederique Oggier
Jean-Baptiste Orfila
Emmanuela Orsini
Tapas Pal
Ying-yu Pan
Roberto Parisella
Aditi Partap
Alain Passelègue
Alice Pellet-Mary
Zachary Pepin
Octavio Perez Kempner
Edoardo Perichetti

Léo Perrin
Naty Peter
Richard Petri
Rafael del Pino
Federico Pintore
Erik Pohle
Simon Pohmann
Guru Vamsi Policharla
Daniel Pollman
Yuriy Polyakov
Alexander Poremba
Eamonn Postlethwaite
Sihang Pu
Luowen Qian
Tian Qiu
Rajeev Raghunath
Srinivasan Raghuraman
Mostafizar Rahman
Mahesh Rajasree
Somindu Chaya Ramanna
Simon Rastikian
Anik Raychaudhuri
Martin Rehberg
Michael Reichle
Krijn Reijnders
Doreen Riepel
Guilherme Rito
Matthieu Rivain
Bhaskar Roberts
Marc Roeschlin
Michael Rosenberg
Paul Rösler
Arnab Roy
Lawrence Roy
Luigi Russo
Keegan Ryan
Markku-Juhani Saarinen
Éric Sageloli
Dhiman Saha
Sayandeep Saha
Yusuke Sakai
Kosei Sakamoto
Subhabrata Samajder
Simona Samardjiska
Maria Corte-Real Santos
Sina Schaeffler
André Schrottenloher
Jacob Schuldt
Mark Schultz
Mahdi Sedaghat
Jae Hong Seo
Yannick Seurin
Aein Shahmirzadi
Girisha Shankar
Yixin Shen
Rentaro Shiba
Ardeshir Shojaeinasab
Jun Jie Sim
Mark Simkin
Jaspal Singh
Benjamin Smith
Yongha Son
Fang Song
Yongsoo Song
Pratik Soni
Pierre-Jean Spaenlehauer
Matthias Johann Steiner
Lukas Stennes
Roy Stracovsky
Takeshi Sugawara
Adam Suhl
Siwei Sun
Elias Suvanto
Koutarou Suzuki
Erkan Tairi
Atsushi Takayasu
Kaoru Takemure
Abdullah Talayhan
Quan Quan Tan
Gang Tang
Khai Hanh Tang
Tianxin Tang
Yi Tang
Stefano Tessaro
Sri AravindaKrishnan Thyagarajan
Yan Bo Ti
Jean-Pierre Tillich
Toi Tomita
Aleksei Udovenko
Arunachalaeswaran V.

Artifact Chair

Rei Ueno	Kyoto University, Japan

Artifact Evaluation Committee

Julien Béguinot	LTCI, Télécom Paris, Institut Polytechnique de Paris, France
Aron Gohr	Independent Researcher
Hosein Hadipour	Graz University of Technology, Austria
Akira Ito	NTT Social Informatics Laboratories, Japan
Haruto Kimura	University of Melbourne, Australia and Waseda University, Japan
Kotaro Matsuoka	Kyoto University, Japan
Florian Mendel	Infineon Technologies, Germany
Hiraku Morita	Aarhus University, University of Copenhagen, Denmark
Prasanna Ravi	Nanyang Technological University, Singapore
Élise Tasso	Tohoku University, Japan

Contents – Part VI

Secure Multiparty Computation

Actively Secure Polynomial Evaluation from Shared Polynomial Encodings

Pascal Reisert[1(✉)], Marc Rivinius[1], Toomas Krips[2], Sebastian Hasler[1], and Ralf Küsters[1]

[1] Institute of Information Security, University of Stuttgart, Stuttgart, Germany
{pascal.reisert,marc.rivinius,
sebastian.hasler,ralf.kusters}@sec.uni-stuttgart.de
[2] University of Tartu, Tartu, Estonia
toomas.krips@ut.ee

Abstract. Many of the currently best actively secure Multi-Party Computation (MPC) protocols like SPDZ (Damgård et al., CRYPTO 2012) and improvements thereof use correlated randomness to speed up the time-critical online phase. Although many of these protocols still rely on classical Beaver triples, recent results show that more complex correlations like matrix or convolution triples lead to more efficient evaluations of the corresponding operations, i.e. matrix multiplications or tensor convolutions. In this paper, we address the evaluation of multivariate polynomials with a new form of randomness: polytuples. We use the polytuples to construct a new family of randomized encodings which then allow us to evaluate the given multivariate polynomial. Our approach can be fine-tuned in various ways to the constraints of applications at hand, in terms of round complexity, bandwidth, and tuple size. We show that for many real-world setups, a polytuples-based online phase outperforms state-of-the-art protocols based on Beaver triples.

Keywords: Multi-party computation · randomized encodings · SPDZ

1 Introduction

Multi-Party Computation (MPC) enables multiple parties to perform computations on private inputs without revealing any information about the inputs apart from what can be deduced from the result. State-of-the-art actively secure MPC protocols, like SPDZ [21,22] and related protocols [5,30,31], follow a two-phase approach, where correlated randomness is precomputed in an offline phase, and later consumed in an online phase to efficiently evaluate a function on private inputs. In this setup, a less efficient offline phase is normally considered acceptable since the offline phase can start well before the input data becomes available. Efficiency in two-phase protocols (and generally in MPC protocols) depends on the number of communication rounds needed and the bandwidth, i.e. the amount of data that has to be transmitted between the parties. Local computations, which can be performed without interaction, are usually considered

K.-M. Chung and Y. Sasaki (Eds.): ASIACRYPT 2024, LNCS 15489, pp. 3–35, 2025.
https://doi.org/10.1007/978-981-96-0938-3_1

less problematic as long as hardware requirements, e.g. memory requirements, remain manageable.

In MPC protocols based on additive secret sharing like SPDZ, addition and multiplication with public values are local operations and therefore fast, while the multiplication of secret values requires interaction and correlated randomness. The most common and widely used form of correlated randomness is classical Beaver triples [6]. The standard approach is to represent a function, e.g. a matrix multiplication, as a series of additions and multiplications and then to use a Beaver triple for each multiplication and to add locally. However, this approach is often not the most efficient choice and for several common operations like matrix multiplication [39,44] or tensor convolutions [14,46] there are by now more efficient actively secure MPC solutions that rely on different forms of correlated randomness like matrix or convolution triples.

Many of these operations like simple field multiplication (Beaver triples), matrix multiplication (matrix triples) and tensor convolution (convolution triples) have in common that they are at most quadratic in the secret inputs. Using this property the protocols achieve a low online communication complexity. Additionally, the quadratic nature can be used in the offline phase, e.g. by using the linear homomorphic structure of lattice-based encryption schemes like BGV [11] to generate the correlated randomness efficiently.

For higher-order operations, like the evaluation of (high-degree) multivariate polynomials, the situation is more difficult, and comparable constructions do not exist. We want to address this problem and present a new actively secure MPC protocol and a suitable new form of correlated randomness called *polytuples*, which speeds up the online evaluation of multivariate polynomials compared to the Beaver triples based approach and still has a reasonably fast offline phase.

We want to briefly describe the high-level idea of our approach. A SPDZ-like online phase has the following characteristics: at the beginning n parties $P_1, \ldots, P_n$ possess (among others) additive shares of the input variables $x_0, \ldots, x_{m-1}$, they perform local computations and communicate until each party P_i has a share $[y]_i$ of the result $y = f(x_0, \ldots, x_{m-1})$ (cf. Sect. 3.2 for the definition of additive shares $[\,\cdot\,]$). To open the result, the parties exchange the $[y]_i$ and locally reconstruct the result $\mathsf{Rec}([y]_1, \ldots, [y]_n) := \sum_{i=1}^{n} [y]_i = y$.[1]

This scheme is, however, by no means the only possible construction. In fact, it is enough for the parties to construct any *randomized encoding* [3,26] of f. A randomized encoding is a set of terms $y_0, \ldots, y_{k-1}$ that depend on the inputs (and some randomness) and a reconstruction algorithm Rec such that $\mathsf{Rec}(y_0, \ldots, y_{k-1}) = f(x_0, \ldots, x_{m-1})$. Additionally, $y_0, \ldots, y_{k-1}$ and Rec are chosen in a way to not leak more information than the actual output $f(x_0, \ldots, x_{m-1})$ (cf. the formal Definition 1). Note that randomized encodings contain the classical SPDZ-setup as the special case $y_i = [y]_i$ for $0 \le i < k = n$ where the parties do almost all of the computation in the interactive phase and only a simple sum in the final reconstruction phase. In particular, the evaluation of a degree

[1] In order to get actively secure protocols, the opening protocols additionally include a MAC check (see our full version [42]. Protocol 5 or [22]).

d multivariate polynomial then requires around $\log_2(d)$ rounds of communication, which might be too much, especially in networks with high latency. It is then advantageous to reduce the round complexity by shifting more of the overall computation into a then more complex reconstruction Rec, since this reconstruction is done locally by each party and therefore nevertheless cheap. Naturally, certain limitations apply to this shift of computation. For example, the size k of the encoding should still remain within practical range for two reasons: (i) For a very large k (e.g. exponential) the local evaluation of Rec might still slow down the overall multi-party computation and (ii) all the encodings $y_0, \ldots, y_{k-1}$ have to be created either by the offline phase or through communication with the other parties and hence a large k increases the bandwidth or the offline runtime.

One of the main contributions of this paper is the construction of a new family of efficient randomized encodings of arbitrary multivariate polynomials f which satisfies these constraints and allows an efficient MPC protocol with only one round of communication. To this end, we follow an iterative approach, where we first construct an encoding $y_0, \ldots, y_{k-1}$ such that each y_l is of degree at most $d_1 < \deg(f)$ in the inputs. We next construct a randomized encoding $(y_{ll'})$ for each y_l of even smaller degree $d_2 < d_1$. The idea is that if the parties have a degree d_2 randomized encoding of each y_l, then they can locally reconstruct all y_l and if they have all y_l then they can reconstruct the result $f(x_0, \ldots, x_{m-1})$. Hence the collection of all $y_{ll'}$ is a degree d_2 randomized encoding of f itself.

While the composition (cf. Lemma 2) of randomized encodings is a well-known result [3], we add a twist. Namely, we construct the encodings $y_{ll'}$ in a way that they can be used in the reconstruction of *multiple* y_l, e.g. $y_{ll'}$ occurs in the randomized encoding of y_l and y_ℓ for some l, ℓ. We prove that the multiple use of such an encoding does not affect the security of the resulting overall randomized encoding of f of degree d_2. Thus we need less encodings (of degree d_2) to construct all y_l and hence $f(x_0, \ldots, x_{m-1})$.

In the next iteration step, we replace the degree d_2 encoding y_l of f by an encoding $y_{ll'l''}$ of even smaller degree $d_3 < d_2$. Again we can find $y_{ll'l''}$ which can be used in the reconstruction of multiple $y_{ll'}$ and we only need to construct a small number of these $y_{ll'}$ by the previous step. Hence iterating further the advantage of our multipurpose encodings becomes more significant and allows us to e.g. construct a degree 3 encoding of $f(x_0, \ldots, x_{m-1}) = x_0 \cdots x_{m-1}$ with output size in $\mathcal{O}(m \log(m))$. Previous results like [18] reached $\mathcal{O}(m^2)$.

In order to use the new randomized encodings to locally reconstruct the results, the parties first need to construct the components y_l in an interactive protocol. We therefore build a new MPC online protocol based on a new form of correlated randomness, i.e. our *polytuples*. Polytuples are specially crafted to allow the computation of the shares $[y_l]$ in only one round of online communication. These shares are then (partly) opened and each party can locally reconstruct the output $f(x_0, \ldots, x_{m-1})$ (or a share thereof).

Our new family of randomized encodings contains a large number of randomized encodings for each single polynomial f. While all these randomized encodings use the aforementioned optimization with multipurpose encodings,

Table 1. Comparison for the computation of $[x_1^{d/m} \cdots x_{m-1}^{d/m}]$ of degree d with $d/m \in \mathbb{N}$, for Beaver triples, binomial tuples, and polytuples.

Approach	Rounds	Bandwidth	Tuple Size
Beaver Triples	$\lceil \log d \rceil$	$2(m-1)\lceil \log \frac{d}{m} \rceil$	$3(m-1)\lceil \log d/m \rceil$
e.g. for $d = m = 16$	4	30	45
Binomial Tuples (see Footnote 2) [15]	1	m	$(\frac{d}{m}+1)^m - 1$
e.g. for $d = m = 16$	1	16	65535
Example Intermediate Polytuple	1	$\mathcal{O}(m \log(m))$	$\mathcal{O}(d(\log m)^2)$
e.g. for $d = m = 16$	1	41	149

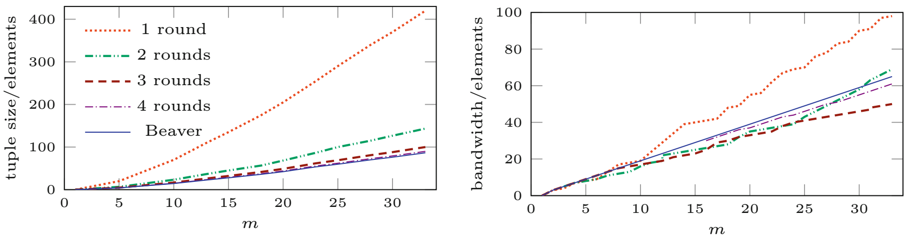


Fig. 1. Multi-round example to evaluate a product of m factors with polytuples with optimal tuple size.

they differ in the number of iteration steps and the degree of the final overall encoding. Moreover, we can use encodings of different degrees for different components, e.g. a degree 4 encoding for y_1 and a degree 3 encoding for y_2.

The choice of a randomized encoding for a given polynomial f and the resulting number and shape of the encodings y_l and of the polytuples, strongly influence various aspects of the online and offline phase for the parties. For example, a low number of iteration steps and/or overall encodings of high degree reduce the output size k. Since all encodings have to be opened this decreases the bandwidth. The tradeoff is a larger tuple size and hence a more complex offline phase (see Sect. 4 for the explicit formulas for tuple size and bandwidth).

Table 1 shows one specific kind of randomized encoding and polytuple. This tuple lies between the linear size for Beaver multiplication and the exponential size of the more straightforward one-round approach from [15,16].[2] It has minimal round complexity and a higher bandwidth cost than the other approaches. Almost all other trade-offs are however possible. The exact relation will be explained in Sect. 4.

[2] To the best of our knowledge no name has been fixed for the [15] underlying correlated randomness—we therefore chose *binomial tuples* to refer to this type of randomness within our paper (cf. also Sect. 3.4 for a definition).

Moreover, our protocol is also composable, i.e. we can write a multivariate polynomial $f(x_0,\ldots,x_{m-1}) = g(g_1(x_0,\ldots x_{m-1}),\ldots,g_j(x_0,\ldots x_{m-1}))$ for multivariate polynomials g, g_l $(1 \leq l \leq j)$ and then compute $[g_l(x_0,\ldots x_{m-1})]$ in the first round with our one-round protocol applied to all g_l and then compute $f(x_0,\ldots,x_{m-1})$ in the second round with the protocol applied to g for inputs $[g_l(x_0,\ldots x_{m-1})]$. This feature adds additional flexibility since it allows us to trade round complexity and bandwidth/tuple size; Fig. 1 illustrates that adding just one round can already make a big difference.

Altogether, we can fine-tune our randomized encodings and polytuples for optimal performance in the concrete setting where the protocol is deployed w.r.t. bandwidth, tuple size, and/or round complexity. For example, if network latency is (moderately) high, we should try to minimize round complexity. Similarly, bandwidth/data rate restrictions imply that one should use polytuples with lower bandwidth. If the runtime of the offline phase, local memory or local computation time are important, striving for small tuple sizes is recommended. Our first experiments show that strategic deployment of polytuples can significantly speedup the performance of the online phase.

Our Contributions. In summary, our contributions are as follows:

– We introduce a new family of randomized encodings for the evaluation of multivariate polynomials as well as suitable correlated randomness, i.e. *polytuples*, to integrate the randomized encodings into a dishonest majority actively secure MPC protocol. Our randomized encodings have the smallest known output size for arbitrary monomials. Our approach evaluates a multivariate polynomial in just one round of online communication plus one opening round.
– We compute the tuple size and bandwidth needed in the online phase for all new randomized encodings and corresponding polytuples. Our tuple size is significantly lower than for existing single-round approaches and also multi-round computations yield improvements (e.g. lower bandwidth and round complexity than Beaver multiplication).
– We evaluate the performance of our approach for sample applications (evaluation of polynomials, comparisons of secret-shared values, simple machine learning algorithms) in Sect. 5 which shows that polytuples speed-up these computations compared to Beaver multiplication.

For further results and details we refer to the full version [42] of this paper.

2 Related Work

We see our work as an improvement over the common online phase of SPDZ [22] and related protocols [5,30,31]. We therefore concentrate our discussion on recent progress applicable to SPDZ-like papers, rather than classical theoretical results like e.g. [17], or other MPC approaches like garbled circuits.

A first small optimization of the Beaver triple-based online phase in SPDZ already appeared in [21] where square pairs are used to improve the squaring of secret shared values. This idea has been picked up by Morten Dahl who describes in [19] power tuples for the computation of a monomial x^d for a secret-shared value x, which are binomial tuples (cf. Sect. 3.4) for a single variable. Dahl [19] also presents matrix triples and convolution triples which have also been discussed in [39] in the passively secure domain, too. Matrix (and convolution) triples have since then seen further attention and are by now available as part of an actively secure protocol [14,25,46]. The multivariate version of binomial tuples appears in the passively secure protocol of [15] with additional trust assumptions on the dealer, whereas the authenticated binomial tuples in this paper provide active security. Ohata and Nuida [40] as well as Couteau [16] use a slight variation of a binomial tuple in the passively secure setup.

Another classical approach to the secure evaluation of a polynomial is included in [4] and again in [20]. The more recent extension presented in [38] uses multiplicative masking. Their combined passively-secure protocols need $4+1+2$ rounds of (online) communication (cf. [13]). The general idea of using a multiplicative structure in the underlying primitives, e.g. a multiplicative secret sharing as in [8,24], is quite tempting. However, these multiplicative sharings generally cannot compute additions in a cheap way, and conversion techniques back to an additive sharing as it is used in SPDZ-like protocols are costly. While these protocols have a constant round complexity and small tuple size, making them actively secure (if possible) usually comes with considerable overhead. Furthermore, there are many papers optimizing the use of maskings/tuples. For example, Boura et al. [9] reuse their masks for certain input variables for different multiplication gates. Moreover, function-dependent preprocessing can be used to decrease the required tuple size and bandwidth in the online phase [7,41]. Also note that with a pseudo-random generator, as in [10], structured randomness can be produced without further communication. Special solutions also exist for more complex structured random data like the matrix triples mentioned before.

Randomized Encodings. Results on randomized encodings reach far back to the works of Ishai and Kushilevitz [26] who proved that every polynomial has a degree-3 randomized encoding. The complexity results of [26] have since been improved by [18] for general branching programs, e.g. for products of m variables they achieve randomized encodings of output size $\mathcal{O}(m^2)$. In comparison, our randomized encoding reduces the output complexity to $\mathcal{O}(m\log(m))$. Other papers like [33] focus on the binary case (which is less related to our arithmetic setup) or relax the correctness or privacy requirements like [3] to achieve better efficiency. We refer to [27] for further classical references on randomized encodings. At the same time [28] presents new actively secure protocols with linear bandwidth and constant round complexity, but with exponential tuple size. Moreover, [16] considers a multi-round approach which improves the bandwidth from linear in the classical Beaver triple-based approach to $\mathcal{O}(m/\sqrt{\log(m)})$. More recently, a new multi-party adapted version of randomized encodings (MPRE) evolved in [1], where preprocessing and the first communication round are more flexible than in our SPDZ-like setup—the latter is (almost completely) restricted to exchange masked inputs $x_j - a_j$ in the first communication round. The MPRE approach has led to new passively secure and actively secure MPC protocols [1,2,35,36]. The currently best actively secure protocol [35] uses Oblivious Linear Evaluation (OLE) correlated randomness, needs two rounds of communication but bandwidth at least cubic in the number of parties n and in $\mathcal{O}(m^{1.5})$ in the online phase. In comparison, our protocols are linear in n and require only $\mathcal{O}(m \log(m))$ communication in the same number of rounds in the online phase.

3 Preliminaries

For our theoretical considerations in Sects. 4.2 to 4.5 we are working on a commutative base ring R. For all other parts, we choose R a finite field as in [22]. We call a computation local if the parties can perform it without interaction.

3.1 Performance Measures

When we analyze the theoretical performance of our protocols, bandwidth is measured in the number of ring elements sent. Analogously, the size of the structured randomness needed for one polynomial evaluation in the online phase, i.e. the tuple size, is the number of ring elements contained in the tuple. The round complexity of a protocol is the number of communication rounds. One communication round consists of all information that can be sent in parallel. In particular, if in a protocol party P_1 has to wait for a message from P_2 before P_1 can send her message, the protocol has round complexity 2. The opening phase in actively secure SPDZ-like protocols comes with an additional invocation of a MAC check subroutine (cf. Sect. 3.2 and [42]. Protocol 7)—to account for the different structures of an opening round we will count opening rounds separately, usually indicated by a "+1" in the round/bandwidth count. It is quite common to ignore the opening round completely for composable protocols since to compute the composition of two or more functions the parties need only one global opening round. E.g. if parties can compute a function f in $k_f + 1$ rounds and function g in $k_g + 1$ rounds, they can compute $g \circ f$ in $k_f + k_g + 1$ rounds. To simplify notation, we sometimes drop the "+1".

3.2 Secret-Sharing and SPDZ-MACs

As we focus on MPC in the dishonest majority setting, we use classical additive secret-sharing, denoted by $[\,\cdot\,]$. A secret x is shared among n parties such that $x = \sum_{i=1}^{n} [x]_i$ where $[x]_i$ is the share of party P_i. All shares are needed to reconstruct a secret and $n-1$ or less shares do not reveal any information. This secret sharing scheme is linear, i.e., we can set $[x+y]_i := [x]_i + [y]_i$, $[cx]_i := c \cdot [x]_i$, $[x+c]_i := [x]_i + c \cdot \delta_{i1}$ for shared values x, y and a publicly known constant c, where δ_{ij} is the Kronecker delta. To open (or reconstruct) a secret-shared value, parties simply broadcast their shares and compute the sum of all shares. Our techniques are independent of the secret-sharing scheme.

In SPDZ and related protocols, shares are additionally authenticated to verify the outputs of the protocol using a MAC key [21,22]. The MAC key $\alpha \in R$ is shared in the preprocessing phase. Secret shared values (including inputs and structured randomness like Beaver triples or polytuples) are authenticated in the offline phase—we use $[\![x]\!] := ([x], [\alpha x])$ to denote authenticated shares of x and $[\![X]\!] = ([\![x_1]\!], \ldots, [\![x_k]\!])$ for a tuple $X = (x_1, \ldots, x_k)$. Linear operations on authenticated shares are a trivial extension of linear operations on shares with the exception of $[\![x + c]\!]_i := ([x + c]_i, [\alpha x]_i + c \cdot [\alpha]_i)$. A MAC check enables parties to verify the integrity of previously opened shares (cf. [42]. Protocol 7 or [21,22]). The soundness of the MAC check is proportional to $\frac{1}{|R|}$ if R is a field, can be aggregated over many opened values, and does not reveal the MAC key [21].

3.3 Randomized Encodings and Randomizing Polynomials

In our protocols we use randomized encodings [26] to reduce the communication rounds, bandwidth, and tuple size.

Definition 1. *Let $X, Y, \hat{Y}, A$ be finite sets and let $f : X \to Y$. A function $\hat{f} : X \times A \to \hat{Y}$ is called* randomized encoding *of f if the following holds:*

- **Correctness.** *There exists a reconstruction algorithm* $\mathsf{Rec} : \hat{Y} \to Y$ *such that* $\mathsf{Rec} \circ \hat{f} = f \circ \mathrm{pr}_1$ *where* $\mathrm{pr}_1 : X \times A \to X, (x, a) \mapsto x$ *is the projection.*
- **Privacy.** *There exists a simulator* Sim *such that* $\mathrm{Sim}(f(x))$ *and* $\hat{f}(x, a)$ *are identically distributed for all $x \in X$ if a is sampled uniformly from A.*

If $\hat{Y} = R^k$, we call the component functions of a randomized encoding, simply encodings *or* randomizing polynomials. *An encoding y_0 of $\hat{f} = (y_l)_{0 \le l < k}$ which is only added by the reconstruction algorithm, i.e.* $\mathsf{Rec}(0, y_1, \ldots, y_{k-1}) + y_0 = \mathsf{Rec}(y_0, y_1, \ldots, y_{k-1})$, *is called* additive.

In this paper, we usually have $X = R^m, Y = R, \hat{Y} = R^k$. The randomness space A is generally more complicated since it is a subvariety of some R^t defined by the structure of our randomness, e.g. for Beaver triples we would choose $A = \{(a, b, c) \in R^3 : ab = c\} \subset R^3$. We remark that for our MPC application, we also include components that are completely deterministic in the other components, e.g. $c = ab$ in the Beaver triple case, since we have to construct this randomness in the offline phase. For possible other applications of our randomized encodings, these deterministic components of A can be omitted.

Moreover, in our arithmetic setup we only need to consider randomized encodings where the entries y_l of $\hat{f}$ are *randomizing polynomials* in $m + t$ variables, i.e. $y_l : X \times A \to R, ((x_j)_{0 \le j < m}, (a_j)_{0 \le j < t}) \mapsto y_l(x_0, \ldots, x_{m-1}, a_0, \ldots, a_{t-1})$ is a polynomial in $x_0, \ldots, x_{m-1}, a_0, \ldots, a_{t-1}$. To simplify the notation we usually drop the explicit dependency of the y_l on x_j and a_j.

A randomized encoding $\hat{f}$ is said to be of *(total) degree-d*, if the entries y_l of $\hat{f}$ are of total degree at most d—both the x_j and the a_j count towards the total degree, e.g. $2x_0a_0^2$ has total degree 3. We write $\hat{f}$ is of x-degree d if it is of degree d in the variables x_j and of a-degree d if it is of degree d in the randomness a_j, i.e. $2x_0a_0^2$ is of x-degree 1 and a-degree 2.

The *output size* of a randomized encoding is the R-rank of $\hat{Y}$, i.e. in this paper the size k. In our protocols, the output size usually coincides (up to an addition by m) with the bandwidth of the corresponding MPC protocol. The *randomness size* t on the other hand corresponds to the tuple size of the employed polytuple.

For later use we recall some fundamental properties for the concatenation and composition of randomized encodings [3]:

Lemma 1. *Let $\hat{f}_i(x, a_i)$ be randomized encodings for $f_i(x)$ with reconstruction algorithm Rec_i and $0 \le i < k$, then $\hat{f}(x, (a_i)_{0\le i<k}) = (\hat{f}_i(x, a_i))_{0\le i<k}$ is a randomized encoding of $f(x) = (f_i(x))_{0\le i<k}$ with reconstruction $\mathsf{Rec} = (\mathsf{Rec}_i)_{0\le i<k}$.*

Lemma 2. *Let $(\hat{f}(x, a), \mathsf{Rec})$ be a randomized encoding of $f(x)$ and $(\hat{f}'((x, a), a'), \mathsf{Rec}')$ a randomized encoding of $\hat{f}(x, a)$ (as a deterministic function of (x, a)). Then $\tilde{f}(x, (a, a')) = \hat{f}'((x, a), a')$ is a randomized encoding of $f(x)$ with reconstruction $\mathsf{Rec} \circ \mathsf{Rec}'$.*

3.4 Binomial Tuples

As mentioned before, our new MPC protocols[3] for the evaluation of a multivariate polynomial f rely on suitable randomized encodings $(y_0, \ldots, y_{k-1})$ of f. Here, the single encodings y_l are built by an interactive one-round protocol that uses structured randomness. Since the y_l might have a degree larger than 2, Beaver triples are not enough and we need a type of structured randomness that allows us to build higher degree terms y_l in one round. The solution is what we call *binomial tuples* (*for* y_l) since their construction is (just like Beaver triples) based on binomial expansion. We want to briefly present binomial tuples and the corresponding MPC online protocol. A passively secure version of this protocol was used in [15,16].

The goal of the binomial tuple approach is to compute a polynomial f in m variables $x_0, \ldots, x_{m-1} \in R$ of total degree $d = \sum_{j=0}^{m-1} d_j$ with one round of communication plus one opening round.

Let $x = (x_0, \ldots, x_{m-1})$ and denote by $f_a(x) = f_{a_0,\ldots,a_{m-1}}(x) = f(x_0 + a_0, \ldots, x_{m-1} + a_{m-1})$ a randomization. As a multi-variate polynomial f_a has the general form $\sum_{e\in E} b_e x^e$ for $x^e := \prod_{j=0}^{m-1} x_j^{e_j}$ and multi-index $e = (e_0, \ldots, e_{m-1}) \in \bigtimes_{j=0}^{m-1}\{0, \ldots, d_j\} =: E$ and some coefficients $b_e \in R$ (which depend on the a_j). Now each party P_i receives (from the offline phase) a share $[\![b_e]\!]_i$ for all $e \in E$. We call the $(b_e)_{e\in E}$ (or the sharing $[\![b_e]\!]_{e\in E}$) a *binomial tuple.*

Additionally, assume that the parties already hold shares $[\![x_j]\!], [\![a_j]\!]$ of the input variables x_j and masks a_j. In the first round of (online) communication

[3] The protocol will be presented later in [42]. Protocol 5.

the parties exchange $[x_j] - [a_j] = [x_j - a_j]$ for $0 \leq j < m$ and reconstruct $x - a = (x_0 - a_0, \dots, x_{m-1} - a_{m-1})$. Subsequently, each party P_i can locally compute a share

$$[\![f(x)]\!]_i = [\![f_a(x-a)]\!]_i = \sum_{e \in E} [\![b_e]\!]_i (x-a)^e \tag{1}$$

i.e. the parties can reconstruct $f(x)$ in the opening round.

Remark 1. If $f(x) = x^{(d_0, \dots, d_{m-1})}$ is a monomial, then

$$f_a(x) = (x+a)^{(d_0,\dots,d_{m-1})} = \sum_{e \in E} \left(\prod_{j=0}^{m-1} \binom{d_j}{e_j} \right) a^{(d_0,\dots,d_{m-1})-e} x^e.$$

Hence, we have $b_e = \left(\prod_{j=0}^{m-1} \binom{d_j}{e_j}\right) a^{(d_0,\dots,d_{m-1})-e}$. Thus, each party needs to receive a share of b_e from the preprocessing, i.e. the tuple size is $\prod_{j=0}^{m-1}(d_j+1)-1$, where the d_j+1 comes from running through the powers 0 to d_j and the final -1 corresponds to the case $e = (d_0, \dots, d_{m-1})$ where $b_e = 1$ is constant and does not have to be shared explicitly. We see that the structured randomness $(b_e)_{e \in E}$ has a small size if $m = 1$, but becomes exponential for monomials of many different factors, e.g. for $d_j = 1$ for all $0 \leq j < m$ one has size $2^m - 1 = 2^d - 1$.

Although binomial tuples come with a minimal round complexity of 1+1 rounds and small bandwidth, e.g. $m+1$ ring elements for the polynomial $\prod_{j=0}^{m-1} x_j$, the often large tuple size makes binomial tuples too inefficient for most higher degree multivariate polynomial evaluations. Our *polytuples* (cf. Definition 2) will therefore not contain binomial tuples for high-degree polynomials, but rather combine and correlate low-degree binomial tuples to retain a small tuple size and bandwidth while keeping the round complexity minimal.

4 Our MPC Protocols for the Evaluation of Multivariate Polynomials

We now present our main technical results on randomized encodings and polytuples. In Sect. 4.1 we explain first what kind of randomized encodings are compatible with our MPC protocol and how they can be used in an online phase. Sections 4.2 to 4.5 construct our new family of suitable randomized encodings. It also analyzes the complexity of the randomized encodings and connects it to the bandwidth and tuple size of our MPC protocols. Finally, Sect. 4.6 contains our MPC protocols and the main theorems. We refer to Sect. 4.7 and [42]. Appendix C for a discussion on the polytuple generation in the offline phase.

4.1 MPC With Randomized Encodings

Our MPC online protocols (just like SPDZ) consider n parties $P_1, \dots, P_n$ that receive shares of the input variables $x = (x_0, \dots, x_{m-1})$ as well as shares of (structured) random data in the form of a structured random tuple $\hat{a} = (a_0, \dots, a_{t-1})$ with $t \geq m$ from an offline phase. The parties can locally add the shares, but they need to interact to compute the product of two secrets like $x_0 x_1$ or $x_0 a_0$. In order to compute these products the parties have to exchange their shares, obviously not in plain, but in some masked form. Therefore, as in SPDZ (and for the binomial tuples in Sect. 3.4) we assume that the parties open $x_j - a_j, 0 \leq j < m$, in an initial round of communication. Thus after the initial communication round, all parties know the public values $x_j - a_j, 0 \leq j < m$, in addition to the shares already provided by the offline and input phase.

The parties can use this information to construct new shares $[y_l]_i$ between the initial communication round and the final opening round. They can locally multiply and add the public values $x_j - a_j$, but they cannot locally multiply the shares (in a meaningful way). Hence the $[y_l]_i$ can be polynomials in the $x_j - a_j$ with coefficients that have at most total degree 1 in $[x_0]_i, \dots, [x_{m-1}]_i, [a_0]_i, \dots, [a_{t-1}]_i$. E.g. $[a_2]_i (x_1 - a_1)^2$ can be computed locally by party P_i after the initial round of communication. However, $[a_1]_i \cdot [a_2]_i (x_1 - a_1)^2 \neq \left[a_1 a_2 (x_1 - a_1)^2\right]_i$, i.e. the degree 2 coefficient $[a_1]_i \cdot [a_2]_i$ is not sufficient to compute a share of the product locally. Instead, we need to include structure randomness $a_3 = a_1 a_2$ in the tuple. Then P_i easily computes $[a_3]_i (x_1 - a_1)^2 = [a_1 a_2]_i (x_1 - a_1)^2 = \left[a_1 a_2 (x_1 - a_1)^2\right]_i$, which now has a coefficient $[a_3]_i$ of degree 1.

After the local computation, the parties open the $[y_l]_i$ and each party gets $y_l = \sum_{i=1}^{n} [y_l]_i$. Note that the degree condition ensures that y_l turns into a polynomial in the x_j and a_j since the shares dissolve, e.g. $\sum_{i=1}^{n} [a_2]_i (x_1 - a_1)^2 = a_2 (x_1 - a_1)^2$. In order to compute $f(x_0, \dots, x_{m-1})$ privately the y_l (together with the $x_j - a_j$) must be a randomized encoding for a suitable reconstruction algorithm Rec, i.e. $\hat{f}(x_0, \dots, x_{m-1}, a_0, \dots, a_{t-1}) = (x_0 - a_0, \dots, x_{m-1} - a_{m-1}, y_0, \dots, y_{k-1})$ in the notation of Definition 1.[4] In particular, the parties can then locally apply Rec to $x_j - a_j$ and the now public y_l, to compute $f(x_0, \dots, x_{m-1})$. Hence for a randomized encoding $\hat{f} = (y_l)_{0 \leq l < k}$ of f with

(I) y_l is a polynomial $\sum_{e(l) \in E(l)} b_{e(l)}(x, \hat{a}) \cdot (x - a)^{e(l)}$ where $E(l) \subset \mathbb{N}^m$ some finite set of multi-indices and $a = (a_0, \dots, a_{m-1})$ the input masks, and
(II) all coefficients $b_{e(l)}(x, \hat{a})$ have total degree at most 1 in $R[x_0, \dots, x_{m-1}, a_0, \dots, a_{t-1}]$,

the parties $P_1, \dots, P_n$ can compute $f(x_0, \dots, x_{m-1})$ with Protocol 1 and option `continuation` = `open`. To later use our randomized encodings in multi-round online protocols (cf. Protocol 1 and [42]. Protocol 5) we furthermore require that

(III) y_0 is an additive component in the sense of Definition 1.

This allows the options `continuation` = share in Protocol 1 below to output a share $[\![f(x_0, \dots, x_{m-1})]\!]_i$ of the result to each party P_i or to output a masked result $f(x_0, \dots, x_{m-1}) - b$ if `continuation` is a shared random value $[\![b]\!]$. We

[4] In our encodings $\hat{f}$ we usually do not include the $x_j - a_j$ explicitly, since we can directly include polynomials in the $x_j - a_j$ in the y_l.

discuss the multi-round use in more detail in Sect. 4.6. The protocol $\Pi_{\text{polynomial}}$ for polynomial evaluations is the core part of our online phase. All other parts, e.g. the input protocol, are identical to their counterparts in SPDZ. We have included the full online protocol Π_{online} in [42]. Protocol 5.

$\Pi_{\text{polynomial}}$

Let $\hat{f} = (y_l)_{0 \le l < k}$ be a randomized encoding of f that satisfies (I), (II), (III) with randomness space A. Each party has a share of $x = (x_0, \ldots, x_{m-1})$ and of some $\hat{a} = (a_0, \ldots, a_{t-1}) \in A$. On input $(\hat{f}, [\![x]\!], [\![\hat{a}]\!], \texttt{continuation})$ each party P_i does:

1. P_i locally computes and then opens $[\![x_j]\!]_i - [\![a_j]\!]_i$ for all $0 \le j < m$. After receiving all shares, P_i locally computes $x_j - a_j$.
2. P_i locally computes $[\![y_l]\!]_i = \sum_{e(l) \in E(l)} b_{e(l)}([\![x]\!]_i, [\![\hat{a}]\!]_i)(x-a)^{e(l)}$ for all $0 \le l < k$, $a = (a_0, \ldots, a_{m-1})$. If $\texttt{continuation} = [\![b]\!]$ then set $[\![y_0]\!]_i \leftarrow [\![y_0]\!]_i - [\![b]\!]_i$.
3. P_i opens $[\![y_l]\!]_i$ for all $i > 0$ and locally computes $y_l = \sum_{i=1}^{n} [y_l]_i$ by summing up the received shares.
   a. If $\texttt{continuation} = \texttt{share}$, P_i locally constructs $[\![f(x_0, \ldots, x_{m-1})]\!]_i = [\![y_0 + \mathsf{Rec}(0, y_1, \ldots, y_{k-1})]\!]_i = [\![y_0]\!]_i + \mathsf{Rec}(0, y_1, \ldots, y_{k-1})\delta_{1i}$.
   b. If $\texttt{continuation} \neq \texttt{share}$, P_i opens and computes $y_0 = \sum_{i=1}^{n} [y_0]_i$ and locally reconstructs $f(x_0, \ldots, x_{m-1}) = \mathsf{Rec}(y_0, \ldots, y_{k-1})$.

Protocol 1. 1(+1) round interactive evaluation of a polynomial f.

Remark 2. As usual for SPDZ-like protocols, we get a passively secure version if we replace $[\![\cdot]\!]$ with a simple $[\cdot]$. We note that all constructions in this paper still work in the passive setup with this modification.

4.2 Our Randomized Encodings

We now want to construct suitable randomized encodings of arbitrary multivariate polynomials compatible with our MPC online phase, i.e. randomized encodings that satisfy (I)–(III) above. We already know from Sect. 3.4 that every multivariate polynomial can be computed with binomial tuples and also that these binomial tuples become too large for high-degree polynomials. Hence we will first construct low-degree randomized encodings and then use the binomial tuples from Sect. 3.4 to construct these low-degree terms as in (1) and Protocol 1, respectively.

We will start with homogeneous monomials $x_{0,\ldots,m-1} := x_0 \cdots x_{m-1}$, then lift our construction to arbitrary monomials, i.e. $x_0^{d_0} \cdots x_{m-1}^{d_{m-1}}$, and finally to arbitrary polynomials.

Idea of Our Construction. To construct a randomized encoding for $f(x_0, \dots, x_{m-1}) = x_{0,\dots,m-1} = \prod_{j=0}^{m-1} x_j$, we follow an iterative approach, where we first construct a degree d_1 encoding $\hat{f}^{(1)}$ of f for some $d_1 < m$, i.e. the components $y_l^{(1)}$ of $\hat{f}^{(1)}$ are polynomials of degree $\leq d_1$. We next construct a lower degree encoding $\hat{f}^{(2)}$ of degree $d_2 < d_1$, of $\hat{f}^{(1)}$ and use the composition Lemma 2 to get a new degree-d_2 encoding of f. Iteratively, we can reduce the degree of the encoding to a target degree, e.g. a degree-3 encoding.

The straightforward approach to construct a randomized encoding of $\hat{f}^{(1)}$ is to construct encodings for each of the component functions $y_l^{(1)}$ of $\hat{f}^{(1)}$ and then to concatenate the encodings with Lemma 1 to a randomized encoding of the whole $\hat{f}^{(1)}$. As mentioned before, in this paper we follow a more efficient approach, where we construct encodings $y_l^{(2)}$ that can be used in the reconstruction of *multiple* $y_l^{(1)}$.

We want to illustrate our approach with the special case $m = 2^n$. We use 3 types of encodings each linear in some monic monomial $x_{u,\dots,r-1} :- x_u \cdots x_{r-1}$ with $u < r$:

(i) with *constant prefactor* 1, i.e. of the form $f_{a_{u,\dots,r-1}}(x_u, \dots, x_{r-1}) = x_{u,\dots,r-1} - a_{u,\dots,r-1}$ and an $a_{u,\dots,r-1} \in A$;
(ii) with *one randomized prefactor* $a \in A$, i.e. of the form $g^a_{b_{u,\dots,r-1}}(x_u, \dots, x_{r-1}) = a x_{u,\dots,r-1} - b_{u,\dots,r-1}$ and a $b_{u,\dots,r-1} \in A$;
(iii) with *two randomized prefactors* $a, b \in A$, i.e. of the form $h^{a,b}_{c_{u,\dots,r-1}}(x_u, \dots, x_{r-1}) = ab x_{u,\dots,r-1} - c_{u,\dots,r-1}$ and a $c_{u,\dots,r-1} \in A$.

We now want to construct randomized encodings for each of these three types of encodings which again consist of terms of type (i), (ii), or (iii), but of lower degree, i.e. with smaller $r - u$. Since our monomial $f(x_0, \dots, x_{m-1})$ is of type (i) with $u = 0, r = m$, this will allow us to construct a degree d_1 encoding $\hat{f}^{(1)} = (y_l^{(1)})$ where all $y_l^{(1)}$ are of type (i), (ii) or (iii) with x-degree $< r$. Then we can iterate.

To simplify notation we use a helper function

$$\varphi(x, y, a, b, c) :- (x - a, y - b, bx + ay - ab - c)$$

on R^5. Moreover, we choose a reconstruction $\mathsf{Rec}(y_0, y_1, y_2) :- y_0 y_1 + y_2$ for output size 3 randomized encodings. Please note that y_2 is then an additive component in the sense of Definition 1. We get

$$\mathsf{Rec} \circ \varphi(x, y, a, b, c) = \varphi_0 \varphi_1 + \varphi_2 = (x - a)(y - b) + bx + ay - ab - c = xy - c. \tag{$*$}$$

Hence, we find for $v = (u + r)/2$ randomized encodings of $f_*, g^a_*, h^{a,b}_*$:

(1) $\hat{f}_{a_{u,\dots,r-1}}(x_u, \dots, x_{r-1}, a_0, a_1) = \varphi(x_{u,\dots,v-1}, x_{v,\dots,r-1}, a_0, a_1, a_{u,\dots,r-1})$[5],

[5] To simplify the notation we often write $*$ for the additive constant index if the index is clear from context.

(2-1) $\hat{g}^a_{b_{u,\dots,r-1}}(x_u,\dots,x_{r-1},b_0,b_1) = \varphi(ax_{u,\dots,v-1}, x_{v,\dots,r-1}, b_0, b_1, b_{u,\dots,r-1})$,

(2-2) $\hat{g}^b_{b_{u,\dots,r-1}}(x_u,\dots,x_{r-1},b_0,b_1) = \varphi(x_{u,\dots,v-1}, bx_{v,\dots,r-1}, b_0, b_1, b_{u,\dots,r-1})$,

(3) $\hat{h}^{a,b}_{c_{u,\dots,r-1}}(x_u,\dots,x_{r-1},c_0,c_1) = \varphi(ax_{u,\dots,v-1}, bx_{v,\dots,r-1}, c_0, c_1, c_{u,\dots,r-1})$,

where $a_0, a_1, b_0, b_1, c_0, c_1 \in A$ are random numbers (not necessarily different). Note that for g we have two different cases depending on whether a randomized prefactor comes from the first component or the second.

While correctness follows in all four cases directly from ($*$), we omit the security proof for now and refer to the general cases discussed in Sect. 4.3.

Please note that in all of these randomized encodings the components (given by some $\varphi_0, \varphi_1, \varphi_2$) are in fact linear combinations of terms of types (i), (ii) or (iii) and of x-degree $(r-u)/2 = 2^{n-1}$. Hence, we can iteratively apply the four randomized encodings again to get to an even smaller x-degree.

The first two components of (1), (2-1), (2-2), (3) (which come from some φ_0, φ_1) are simple terms of types (i)–(iii). For these we can iterate immediately, i.e. apply (1), (2-1), (2-2), (3) with either $u \leftarrow u, r \leftarrow v, v \leftarrow (u+r)/2$ or $u \leftarrow v, r \leftarrow r, v \leftarrow (u+r)/2$ to get encodings of the components of x-degree $(r-u)/2 = 2^{n-1}$ and output size 3. In the third components (corresponding to φ_2) we have sums of type (i)–(iii) terms. Here, we construct a randomized encoding for each summand (using suitable instances of (1), (2-1), (2-2), (3)) and then combine them to a randomized encoding of the sum.[6] Overall, this leads to four randomized encodings of output size 3 (as above) and x-degree 2^{n-1}: two for the first two components and two for the two summands of the third component. If we follow this path and reduce the x-degree iteratively by a factor 2 in each round, then we quickly see that we get (using concatenation Lemma 1 and composition Lemma 2) an overall randomized encoding of x-degree 1 (and a-degree ≤ 2) of output size in $\mathcal{O}(4^n) = \mathcal{O}(m^2)$ similar to the results in [18].

However, if we investigate our randomized encodings above a bit closer, then we see that we produce a significant amount of identical encodings multiple times. For example, if we set $a_1 = b_1$ then the second component of both $\hat{f}_*(x_u,\dots,x_{r-1},a_0,a_1)$ and $\hat{g}^a_*(x_u,\dots,x_{r-1},b_0,a_1)$ is $x_{v,\dots,r-1} - a_1$. Analogously, we get a joined component for (1) and (2-2) if $a_0 = b_0$. Similarly, we see that for $b_0 = c_0$ the first components of both (2-1) $\hat{g}^a_*(x_u,\dots,x_{r-1},b_0,b_1)$ and (3) $\hat{h}^{a,b}_*(x_u,\dots,x_{r-1},b_0,c_1)$ are identical: $ax_{u,\dots,v-1} - b_0$. Analogously, for (2-2) and (3) for $b_1 = c_1$. See also Fig. 2. Thus, if we choose the randomness suitably, it is enough to produce some of the encodings in (1), (2-1), (2-2), (3) only once and then use them in *multiple* reconstructions. E.g. this allows us to save 4 components when constructing a randomized encoding of $(f_*(x_u,\dots,x_{r-1}), g^a_*(x_u,\dots,x_{r-1}), g^b_*(x_u,\dots,x_{r-1}), h^{a,b}_*(x_u,\dots,x_{r-1}))$. Please note that while in general one cannot use the same encoding in different reconstructions without losing privacy, our construction allows the multiple use of encodings—we refer to Corollary 1 for the formal result. We can now conclude that we need for a randomized encoding of

[6] We omit details for this combination, which is treated in general in Corollary 1.

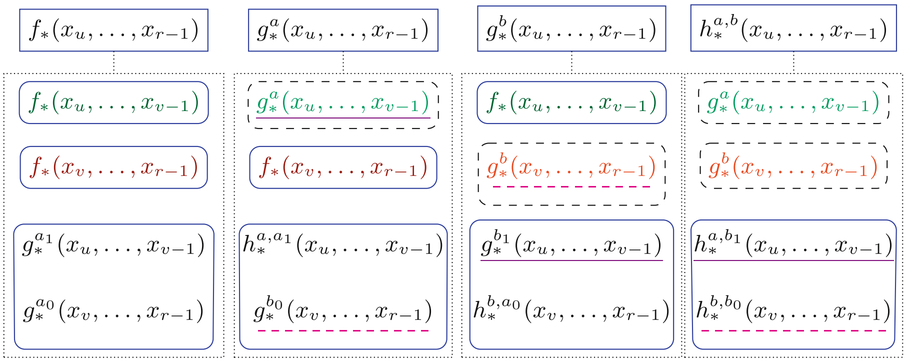

$f_*(x_u,\dots,x_{r-1})$
$f_*(x_u,\dots,x_{v-1})$
$f_*(x_v,\dots,x_{r-1})$
$g_*^{a_1}(x_u,\dots,x_{v-1})$
$g_*^{a_0}(x_v,\dots,x_{r-1})$
$g_*^{a}(x_u,\dots,x_{r-1})$
$g_*^{a}(x_u,\dots,x_{v-1})$
$f_*(x_v,\dots,x_{r-1})$
$h_*^{a,a_1}(x_u,\dots,x_{v-1})$
$g_*^{b_0}(x_v,\dots,x_{r-1})$
$g_*^{b}(x_u,\dots,x_{r-1})$
$f_*(x_u,\dots,x_{v-1})$
$g_*^{b}(x_v,\dots,x_{r-1})$
$g_*^{b_1}(x_u,\dots,x_{v-1})$
$h_*^{b,a_0}(x_v,\dots,x_{r-1})$
$h_*^{a,b}(x_u,\dots,x_{r-1})$
$g_*^{a}(x_u,\dots,x_{v-1})$
$g_*^{b}(x_v,\dots,x_{r-1})$
$h_*^{a,b_1}(x_u,\dots,x_{v-1})$
$h_*^{b,b_0}(x_v,\dots,x_{r-1})$

Fig. 2. Components in the encodings (1) [left], (2-a) [left-middle], (2-b) [right-middle], (3) [right]. Identical colors (apart from black) mark identical encodings, black/dashed boxed components are duplicates and therefore not produced again.

(a) $f_*(x_u,\dots,x_{r-1})$: 2 type (i) terms (1st, 2nd component of $\hat{f}$) and 2 type (ii) terms (summands in the 3rd component of $\hat{f}$),

(b) each $g_*^a(x_u,\dots,x_{r-1})$, $g_*^b(x_u,\dots,x_{r-1})$: 2 additional type (ii) terms (1st (2-1) or 2nd (2-2) component of $\hat{g}$ + one summand in the 3rd component) plus 1 type (iii) term (summand in the 3rd component).

(c) $h_*^{a,b}(x_u,\dots,x_{r-1})$: 2 additional type (iii) terms (summands in the 3rd component of $\hat{h}$).

Please note that (b) assumes that (a) has been already produced; (c) assumes that both (a) and (b) have been produced. Fortunately, this is the only case that occurs in our iterative construction, i.e. whenever we need to construct a term $h_*^{a,b}$ we also need to construct the corresponding g_*^a, g_*^b, f_* linear in the same monomial. Analogous, whenever we need to construct a g_*^a or g_*^b we also need to construct an f linear in the same monomial.

We want to briefly look at two successive iteration steps to explain why this is the case. We start with our monomial $f(x_0,\dots,x_{m-1}) = x_{0,\dots,m-1}$. Then the randomized encoding $\hat{f}_*(x_0,\dots,x_{m-1},a_{0,\dots,2v-1},a_{2v,\dots,m-1})$ for $v = 2^{n-2}, r = m = 2^n, u = 0$ from (1) leads to

– 2 terms $f_*(x_0,\dots,x_{2v-1})$, $f_*(x_{2v},\dots,x_{m-1})$ and 2 terms $g_*^{a_{2v,\dots,m-1}}(x_0,\dots,x_{2v-1})$, $g_*^{a_{0,\dots,2v-1}}(x_{2v},\dots,x_{m-1})$ in the 3rd component of $\hat{f}$ (see also left column in Fig. 2).

If we go one iteration further, i.e. apply (1), (2-1), (2-2), (3) to these four terms, $\hat{f}_*(x_0,\dots,x_{2v-1},a_{0,\dots,v-1},a_{v,\dots,2v-1})$ leads again to 2 terms $f_*(x_0,\dots,x_{v-1}), f_*(x_v,\dots,x_{2v-1})$
and 2 terms $g_*^{a_{v,\dots,2v-1}}(x_0,\dots,x_{v-1}), g_*^{a_{0,\dots,v-1}}(x_v,\dots,x_{2v-1})$. But now we also get from $\hat{g}_*^{a_{2v,\dots,m-1}}(x_0,\dots,x_{2v-1},b_{0,\dots,v-1},a_{v,\dots,2v-1})$ in the case (2-1):[7]

[7] Analogously for (2-2).

– 2 additional terms $g_*^{a_{2v,\ldots,m-1}}(x_0,\ldots,x_{v-1})$, $g_*^{b_{0,\ldots,v-1}}(x_v,\ldots,x_{2v-1})$, and one term $h_*^{a_{v,\ldots,2v-1},a_{2v,\ldots,m-1}}(x_0,\ldots,x_{v-1})$ (see Fig. 2, right-middle column).

We see that we get in fact the 4 terms $f_*, g_*^{a_{v,\ldots,2v-1}}, g_*^{a_{2v,\ldots,m-1}}$, $h_*^{a_{v,\ldots,2v-1},a_{2v,\ldots,m-1}}$ all linear in $x_{0,\ldots,v-1}$. Furthermore, observe that each type (ii) term only occurs with a corresponding type (i) term linear in the same monomial and that each $h_*^{a,b}$ only occurs with corresponding g_*^a, g_*^b and f_* terms all linear in the same monomial. Finally note that a type (iii) term $h_*^{a,b}(x_u,\ldots,x_{r-1})$ again leads to two type (iii) terms $h_*^{a,b_1}(x_u,\ldots,x_{v-1}), h_*^{b,b_0}(x_v,\ldots,x_{r-1})$ (Fig. 2, right). As we have seen, we also have $g_*^a(x_u,\ldots,x_{r-1}), g_*^b(x_u,\ldots,x_{r-1})$. Then we have to apply (2-1) to $g_*^a(x_u,\ldots,x_{r-1})$ and (2-2) to $g_*^b(x_u,\ldots,x_{r-1})$ (or vice versa) to ensure that $g_*^a(x_u,\ldots,x_{v-1})$ and $g_*^b(x_v,\ldots,x_{r-1})$ (or $g_*^b(x_u,\ldots,x_{v-1})$ and $g_*^a(x_v,\ldots,x_{r-1})$) are already available for the reconstruction of $h_*^{a,b}(x_u,\ldots,x_{r-1})$. The two different cases are (dashed) underlined in Fig. 2.

Overall we see that the number of needed encodings as described by (a), (b), (c) hold generally in our construction. Hence we can deduce the output complexity of our iterative approach, namely:

$$\begin{tikzcd}
f & 1 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 2"] & 2 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 2"] & 4 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 2"] & 8 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 2"] & 16 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 2"] & 32 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 2"] & 64 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 2"] & \cdots \\
g & - & 2 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 1"] & 8 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 1"] & 24 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 1"] & 64 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 1"] & 160 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 1"] & 384 \arrow[r, "\cdot 2"] \arrow[dr, "\cdot 1"] & \cdots \\
h & - & - & 2 \arrow[r, "\cdot 2"] & 12 \arrow[r, "\cdot 2"] & 48 \arrow[r, "\cdot 2"] & 160 \arrow[r, "\cdot 2"] & 480 \arrow[r, "\cdot 2"] & \cdots
\end{tikzcd}$$

We easily see that the number of type (i) terms f is in $\mathcal{O}(2^n) = \mathcal{O}(m)$, of type (ii) terms g is in $\mathcal{O}(2^n n) = \mathcal{O}(m\log(m))$ and type (iii) terms h is in $\mathcal{O}(2^n n^2) = \mathcal{O}(m(\log(m))^2)$. Hence we get overall complexity $\mathcal{O}(m(\log(m))^2)$ since we have to construct all of these terms. This is already a significant improvement over the currently best-known result $\mathcal{O}(m^2)$ [18].

However, the result is not ideal yet. We can further improve it by combining additive components: Consider $\mathsf{Rec}'(y_0,y_1,y_2,y_3,y_4) = y_0y_1 + y_2y_3 + y_4$ and

$$\begin{aligned}
&\varphi'(x,y,a,b,x',y',a',b',c)\\
&:= (x-a, y-b, x'-a', y'-b', bx+ay+b'x'+a'y'-ab-a'b'-c)
\end{aligned}$$

Then $\mathsf{Rec}' \circ \varphi' = xy + x'y' - c$. Thus for $v = r/4$:

$$\begin{aligned}
&\hat{f}_{\mathsf{add}}(x_0,\ldots,x_{r-1},a_{0,\ldots,v-1},a_{v,\ldots,2v-1},a_{2v,\ldots,3v-1},a_{3v,\ldots,r-1})\\
&= \varphi'(x_{0,\ldots,v-1}, a_{2v,\ldots,r-1}x_{v,\ldots,2v-1}, a_{0,\ldots,v-1}, b_{v,\ldots,2v-1}, x_{2v,\ldots,3v-1},\\
&\qquad a_{0,\ldots,v-1}x_{3v,\ldots,r-1}, a_{2v,\ldots,3v-1}, b_{3v,\ldots,r-1}, a_{0,\ldots,2v-1}a_{2v,\ldots,r-1})
\end{aligned}$$

is a randomized encoding of the additive 3rd component of $\hat{f}_*$ (from (1)), i.e. of $a_{2v,\ldots,r-1}x_{0,\ldots,2v-1} + a_{0,\ldots,2v-1}x_{2v,\ldots,r-1} - a_{0,\ldots,2v-1}a_{2v\ldots,r-1}$. Note that $\hat{f}_{\mathsf{add}}$ only has 5 components compared to 6 that are needed if we construct each summand separately. Analogous results hold for the additive components of $\hat{g}_*^a, \hat{g}_*^b, \hat{h}_*^{a,b}$. Overall this reduces the output complexity down to $\mathcal{O}(m\log(m))$.

4.3 Technical Lemmas and Formal Results

We now want to present a generalization of the previous construction. The proofs to all statements in this section are available in [42]. Appendix A. We also refer to [42]. Appendix A for additional examples, e.g. [42]. Examples 1, 2, 4 and 5.

We start with the main technical Lemmas 3 to 5. The three lemmas discuss the three cases (i)–(iii) already presented above, i.e. no (Lemma 3), one (Lemma 4) or two randomized prefactors (Lemma 5). While in the previous special case the randomized encoding of $x_{0,\dots,m-1}$ consisted of terms either linear in $x_{0,\dots,t-1}$ or in $x_{t,\dots,m-1}$, the more general lemmas instead allow to construct a randomized encoding linear in any number $1 \le r_1 \le m$ of monomials $x_{S_{1,j}} := \prod_{k \in S_{1,j}} x_k$ for $\{0,\dots,m-1\} = \dot{\bigcup}_{j \in \mathbb{Z}_{r_1}} S_{1,j}$ any disjoint union.[8] E.g. we can split $x_{0,\dots,8}$ into terms linear in the three monomials $x_{0,\dots,2}, x_{3,\dots,5}$ or $x_{6,\dots,8}$.

We will first state the lemmas and then explain how they can be combined into a low-degree encoding of any product $x_0 \cdots x_{m-1}$. Since we later apply the lemmas several times in different degrees, they are stated in a generic degree r instead of m to avoid confusion. Furthermore, we use the following notation: For $\emptyset \neq J = \{j_0,\dots,j_s\} \subset \mathbb{Z}_r$ with representatives $0 \le j_0 < \cdots < j_s < r, j_{s+1} := j_0$ and a set of functions $\{f_{ij}, (i,j) \in \mathbb{Z}_r^2\}$, define the product $f_J := \prod_{v=0}^{s} f_{j_v, j_{v+1}-1}$. E.g. the set $J = \mathbb{Z}_5$ leads to $f_J = f_{0,0} f_{1,1} \cdots f_{4,4}$ and $J = \{2,4,5\} \subset \mathbb{Z}_6$ to $f_J = f_{2,3} f_{4,4} f_{5,1}$.

Lemma 3. *Let $f(x_0,\dots,x_{r-1}) = x_{0,\dots,r-1} - c$ for some consant $c \in R$. There is a randomized encoding $\hat{f}$ with randomness and output size both $r(r-1)+1$. The randomized components of $\hat{f}$ have the form $f_{ii} = x_i - a_i$, and $f_{ij} = x_i a_{i+1,\dots,j} - a_{i,\dots,j}$, and $f_{\mathsf{add}} = \sum_{i \in \mathbb{Z}_r} x_i a_{i+1,\dots,i-1} - a_{\mathsf{add}}$ for randomness $a_i, a_{i+1,\dots,j}$ for $i \neq j \neq i-1$ and $i,j \in \mathbb{Z}_r$, and $a_{\mathsf{add}} = c + \sum_{J \subset \mathbb{Z}_r, |J|>1} (-1)^{|J|} a_J$ where $a_J := \prod_{v=0}^{s} a_{j_v,\dots,j_{v+1}-1}$. The reconstruction function has the form $\mathsf{Rec}(f_{ij}, f_{\mathsf{add}}) := f_{\mathsf{add}} + \sum_{J \subset \mathbb{Z}_r, |J|>1} f_J$.*

Proof. We first note that there are exactly $r^2 - r$ different factors in products f_J associated with sets J with $|J| > 1$, since a factor is defined by its start j_t and end index j_{t+1}, i.e. 2 ordered samples from $\mathbb{Z}_r$ drawn without replacement. We show first that $\prod_{j \in \mathbb{Z}_r} x_j - c = f_{\mathsf{add}} + \sum_{J \subset \mathbb{Z}_r, |J|>1} f_J := \mathsf{Rec}(f_{ij}, f_{\mathsf{add}})$ for a suitable structured randomness a_{add} (constant in the x_j).[9] Note that apart from $\prod_{j \in \mathbb{Z}_r} x_j$ each non-constant summand in the expression on the right is of the form $x_j a_{j+1,\dots,k-1} g$ for some specific term g and some j,k.[10] Each of these terms (for a fixed g, j and k) occurs exactly once with a positive sign for a J which contains $j_l = j, j_{l+1} = k \neq j+1$ for some l, i.e. as a summand in $f_{j_l,k-1} g = (x_{j_l} a_{j_l+1,\dots,k-1} - a_{j_l,\dots,k-1}) g$ or f_{add} if $k = j$.[11] It occurs

[8] We use indices in $\mathbb{Z}_r$ because they wrap around nicely. To be more formal, we will sometimes use $\bar{i}$ for the unique representative of $i \in \mathbb{Z}_r$ in $\{0,\dots,r-1\}$.

[9] We remark that the sum is exponential in r. We will however usually use r small enough that this local computation does not affect the overall runtime significantly.

[10] Take $j := \min\{\bar{i} : x_i a_{i+1,\dots,k}$ a factor of the summand for some $k \neq i+1\}$.

[11] The other elements of J are uniquely determined by g.

exactly once with a negative sign for a $J' = J \cup \{j+1\}$, i.e. as a summand in $f_{j_l,j_l} f_{j_l+1,k-1} g = (x_{j_l} - a_{j_l})(x_{j_l+1} a_{j_l+2,\dots,k-1} - a_{j_l+1,\dots,k-1})g$. Thus these terms cancel out. It remains only $\prod_{j \in \mathbb{Z}_r} x_j$ and constant random terms (in the x_j) which add up to $-c$ for a suitably chosen (structured) randomness a_{add}. Namely, $a_{\mathsf{add}} = c + \sum_{J \subset \mathbb{Z}_r, |J|>1} f_J(0,\dots,0)$ if we consider f_J as a function of the x_i. This shows the correctness of the randomized encoding.

For privacy, we first choose uniformly random (and in particular mutually independent) $a_{i,\dots,j} \in R$ for $i \neq j+1$ and only a_{add} structured, i.e. a (deterministic) polynomial in the $a_{i,\dots,j}$. Now the simulator samples its first $r(r-1)$ components $\tilde{f}_{ij}$ (corresponding to the f_{ij}) uniformly from R. Since each f_{ij} contains an additive random mask (and all masks are independent), the f_{ij} are also distributed uniformly if the $a_{i,\dots,j}$ are sampled uniformly (cf. Definition 1). For the last component $\tilde{f}_{\mathsf{add}}$ (corresponding to f_{add}), the simulator computes $\tilde{f}_{\mathsf{add}} = -\mathsf{Rec}(\tilde{f}_{ij}, 0) + f(x_0,\dots,x_{m-1})$. By construction $f_{\mathsf{add}} = -\mathsf{Rec}(f_{ij}, 0) + f(x_0,\dots,x_{m-1})$ and $\tilde{f}_{\mathsf{add}}$ are equally distributed, which shows privacy. □

Remark 3. Observe that r of the encodings have constant leading coefficient 1 as a polynomial in $x_0,\dots,x_{r-1}$, i.e. the f_{ii}. Moreover, there are $r(r-2)+1$ encodings where the leading coefficient is one random element, i.e. the f_{ij} for $i \neq j \neq i-1$ and f_{add}.

Lemma 4. *Let $g^a(x_0,\dots,x_{r-1}) = a x_{0,\dots,r-1} - c$ for some $a, c \in R$. Let $\mu \in \mathbb{Z}_r$ be a fixed index and define $T_\mu := \{(i,j) \in \mathbb{Z}_r^2 : \overline{j-\mu} \leq \overline{i-\mu-1}\}$ and $S_\mu = \mathbb{Z}_r^2 \setminus T_\mu$. Let $f_{ij}, a_{ij}, \mathsf{Rec}$ be as in Lemma 3. Then there is a randomized encoding $\hat{g}^{a,\mu}$ of g^a with randomness and output size both $r(r-1)+1$. The randomized components of $\hat{g}^{a,\mu}$ have the form*

(i) $g_{ij}^{a,\mu} = f_{ij}$ *for* $(i,j) \in S_\mu$.
(ii) $g_{\mu\mu}^{a,\mu} = a x_\mu - b_\mu^{a,\mu}$, $g_{\mu j}^{a,\mu} = a x_\mu a_{\mu+1,\dots,j} - b_{\mu,\dots,j}^{a,\mu}$ *for* $j \neq \mu, \mu-1$.
(iii) $g_{ij}^{a,\mu} = x_i b_{i+1,\dots,j}^{a,\mu} - b_{i,\dots,j}^{a,\mu}$ *for* $(i,j) \in T_\mu \setminus (\{\mu\} \times \mathbb{Z}_r)$ *and* $j \neq i-1$.
(iv) $g_{\mathsf{add}}^{a,\mu} = a x_\mu a_{\mu+1,\dots,\mu-1} + \sum_{i \in \mathbb{Z}_r \setminus \{\mu\}} x_i b_{i+1,\dots,i-1}^{a,\mu} - b_{\mathsf{add}}^{a,\mu}$.

for randomness $b_{i,\dots,j}^{a,\mu}$ *for* $(i,j) \in T_\mu$ *and* $j \neq i-1$, *and* $b_{\mathsf{add}}^{a,\mu} = c + \sum_{J \subset \mathbb{Z}_r, |J|>1} (-1)^{|J|} b_J^{a,\mu}$ *where* $b_{i,\dots,j}^{a,\mu} := a_{i,\dots,j}$ *for* $(i,j) \in S_\mu$.

Proof. We can simply copy the proof of Lemma 3 for the variables $(a x_\mu, x_j, j \neq \mu)$ and coefficients $b_*^{a,\mu}$ instead of a_*. Please note that again we have to choose the $b_{i,\dots,j}^{a,\mu}$ uniformly random from R and only $b_{\mathsf{add}}^{a,\mu}$ is structured. □

Remark 4. First note that we use the additional index μ to determine to which encoding the prefactor a is assigned. Moreover, observe that $r-1$ of the new terms have as leading coefficient a product of two random elements, i.e. the $a x_\mu a_{\mu+1,\dots,j}$ in $g_{\mu j}^{a,\mu}, g_{\mathsf{add}}^{a,\mu}$ for $j \neq \mu$. The other new encodings all have one random prefactor: $|T_\mu| - r$ encodings from (ii), (iii) as well as $r-1$ summands in $g_{\mathsf{add}}^{a,\mu}$.

Lemma 5. *Let $h^{a,b}(x_0,\ldots,x_{r-1}) = abx_{0,\ldots,r-1} - c$ for some $a,b,c \in R$. Let $\mu,\nu \in \mathbb{Z}_r$ be two fixed indices with $\mu \neq \nu$. Let f_{ij}, $g^{a,\mu}_{ij}$, $g^{b,\nu}_{ij}$, $b^{a,\mu}_{ij}$, $b^{b,\nu}_{ij}$, S_μ, S_ν, T_μ, T_ν,* Rec *be as in Lemmas 3 and 4. Then there is a randomized encoding $\hat{h}^{a,b}$ of $h^{a,b}$ with randomness and output size both $r(r-1)+1$. The randomized components of $\hat{h}^{a,b}$ have the form:*

(i) $h_{ij} = f_{ij}$ *for* $(i,j) \in S_\mu \cap S_\nu$
(ii) $h_{ij} = g^{a,\mu}_{ij}$ *for* $(i,j) \in T_\mu \setminus T_\nu$, $h_{ij} = g^{b,\nu}_{ij}$ *for* $(i,j) \in T_\nu \setminus T_\mu$
(iii) $h_{\mu j} = ax_\mu b^{a,\mu}_{\mu+1,\ldots,j} - c_{\mu,\ldots,j}$ *for* $(\mu,j) \in T_\nu, j \neq \mu, \mu-1$; $h_{\nu j} = bx_\nu b^{b,\nu}_{\nu+1,\ldots,j} - c_{\nu,\ldots,j}$ *for* $(\nu,j) \in T_\mu, j \neq \nu, \nu-1$;
(iv) $h_{ij} = x_i c_{i+1,\ldots,j} - c_{i,\ldots,j}$ *for* $\mu \neq i \neq \nu$ *and* $(i,j) \in T_\mu \cap T_\nu$ *and* $j \neq i, i-1$
(v) $h_{\mathsf{add}} = ax_\mu b^{a,\mu}_{\mu+1,\ldots,\mu-1} + bx_\nu b^{b,\nu}_{\nu+1,\ldots,\nu-1} + \sum_{i\in\mathbb{Z}_r\setminus\{\mu,\nu\}} x_i c_{i+1,\ldots,i-1} - c_{\mathsf{add}}$

for randomness $c_{i,\ldots,j}$ for $(i,j) \in T_\mu \cap T_\nu \wedge (j \neq i-1)$ and $c_{\mathsf{add}} = c + \sum_{J\subset\mathbb{Z}_r,|J|>1}(-1)^{|J|}c_J$ where $c_{i,\ldots,j} :- a_{i,\ldots,j}$ for $(i,j) \in S_\mu \cap S_\nu$, $c_{i,\ldots,j} :- b^{a,\mu}_{i,\ldots,j}$ for $(i,j) \in T_\mu \setminus T_\nu$, $c_{i,\ldots,j} :- b^{b,\nu}_{i,\ldots,j}$ for $(i,j) \in T_\nu \setminus T_\mu$.

Proof. Note that we can in fact consistently set $c_{i,\ldots,j} = a_{i,\ldots,j}$ for $(i,j) \in S_\mu \cap S_\nu$, since then $(i+1,j) \in S_\mu \cap S_\nu$ if $i \neq j$. Set $c_{i,\ldots,j} = b^{a,\mu}_{i,\ldots,j}$ for $(i,j) \in T_\mu \setminus T_\nu$, since then $(i+1,j) \in T_\mu \setminus T_\nu$ apart from $i \neq \mu$. Analogously $c_{i,\ldots,j} = b^{b,\nu}_{i,\ldots,j}$ for $(i,j) \in T_\nu \setminus T_\mu$. Furthermore, $(i,j) \in T_\mu \cap T_\nu \Rightarrow (i+1,j) \in T_\mu \cap T_\nu$ for $\mu \neq i \neq \nu$. In particular, $(i,i-1) \in T_\mu \cap T_\nu$. The claim now follows as in Lemma 3 with variables $(ax_\mu, bx_\nu, x_j : \mu \neq j \neq \nu)$ and randomness c_* instead of a_*. □

Remark 5. Observe that the two indices μ and ν are again used to assign the two prefactors a and b to the encodings linear in x_μ and in x_ν. Moreover, note that the number of new terms with two randomized prefactors is r, i.e. the terms $h_{\mu j}$ for $\overline{j-\nu} \leq \overline{\mu-\nu-1}$ and $h_{\nu j}$ for $\overline{j-\mu} \leq \overline{\nu-\mu-1}$. All other new encodings and summands thereof have one variable prefactor.

Remark 6. Please also note that in the previous lemmas, we always get one (unstructured) random number for each new component apart from the additive component. For the additive component, we get one structured random number.

The previous technical lemmas are combined as in the special case in Sect. 4.1. Namely, we partition our variables $x_0,\ldots,x_{m-1}$ into $r_1 \leq m$ sets, i.e. we choose a partition $\{0,\ldots,m-1\} -: S_{0,0} = \dot\bigcup_{i\in\mathbb{Z}_{r_1}} S_{1,i}$ and consider monomials $x_{S_{1,i}} = \prod_{j\in S_{1,i}} x_j$. Obviously we have $f(x_0,\ldots,x_{m-1}) = x_0\cdots x_{m-1} = \prod_{j\in\mathbb{Z}_{r_1}} x_{S_{1,i}}$. Hence we can apply Lemma 3 with $r \leftarrow r_1, x_i \leftarrow x_{S_{1,i}}$. We receive encodings $(f^{(1)}_{ij}, f^{(1)}_{\mathsf{add}})$ which are linear in the $x_{S_{1,i}}$. Some of these encodings have no randomized leading coefficient (e.g. the $f^{(1)}_{ii}$). For these terms, we can apply Lemma 3 again by partitioning $S_{1,i}$ into smaller sets. For terms with one randomized leading coefficient like the $f^{(1)}_{ij}$ $(i \neq j \neq i-1)$ we analogously apply Lemma 4. By repeatedly applying the Lemmas 3 to 5 we then get encodings linear in some target elementary monomials $x_{S_{\ell,i}}$.

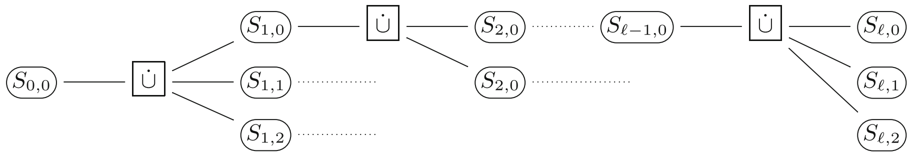


Fig. 3. Tree-like structure of a series of refinements of partitions.

Formally, this approach corresponds to a series of refinements $S_{0,0} = \dot\bigcup_{i\in\mathbb{Z}_{r_k}} S_{k,i}$ of disjoint unions of non-empty sets for $1 = r_0 < r_1 < \cdots < r_\ell \le m$, i.e. $\forall 0 < k \le \ell\ \forall i \in\ \mathbb{Z}_{r_k}\ \exists i_0 \in \mathbb{Z}_{r_{k-1}} : S_{k,i} \subseteq S_{k-1,i_0}$. We get a tree structure visualized in Fig. 3 where $I_{k,i} \coloneqq \{j \in \mathbb{Z}_{r_{k+1}} : S_{k+1,j} \subseteq S_{k,i}\}$ is the number of children of $S_{k,i}$. To later map the indices of these refinements to the generic indices in the lemmas, we fix a bijective map $\psi_{ki} : \mathbb{Z}_{|I_{k,i}|} \to I_{k,i}$ for all $0 \le k \le \ell$ and $0 \le i < r_k$.

In terms of these general refinements, our construction (so far) defines for each monomial $x_{S_{k,i}}$ that occurs in the construction, a randomized encoding linear in the $x_{S_{k+1,j}}$ for $j \in I_{k,i}$. Now in order to combine these single randomized encodings into a randomized encoding of the whole $f(x_0,\ldots,x_{m-1}) = x_0\cdots x_{m-1}$ we need to use concatenation and composition as described before. However, the classical concatenation Lemma 1 assumes independent randomized encodings to be concatenated. In contrast, our constructions in Lemmas 3 to 5 use the same encodings, e.g. the f_{ij} in Lemma 3 and in Lemma 4 (i), for different components. Fortunately, for the encodings that occur in Lemmas 3 to 5 this still leads to a secure concatenation, e.g. $((f_{ij})_{i-1\neq j}, f_{\mathsf{add}}, (g_{ij}^{\mu,a})_{(i,j)\notin S_\mu}, g_{\mathsf{add}}^{\mu,a})$ is a randomized encoding of the concatenation $(f, g^a) = (x_0\cdots x_{r-1}, ax_0\cdots x_{r-1})$.[12] Formally, this property is described by:

Corollary 1. *Let f, g be two functions. Let $\hat f$ be a randomized encoding of f with additive component $\hat f_0$ and simulator Sim_f. Furthermore, let $\hat g$ be a randomized encoding of g with additive component $\hat g_0$ and simulator Sim_g. Assume that for all $i, j > 0$ $(\mathsf{Sim}_f)_i$ and $(\mathsf{Sim}_g)_j$ are independent uniformly random numbers. Let $J = \{j > 0 | \exists i > 0 : \hat f_i = \hat g_j\}$. Then $((\hat f_i)_{0\le i}, (\hat g_j)_{j\notin J})$ is a randomized encoding of $(\hat f, \hat g)$ with output size $k+k'-|J|$. Moreover, if $\hat f_0, \hat g_0$ map to the same (additive) group then $((\hat f_i)_{0<i}, (\hat g_j)_{j\notin J\cup\{0\}}, \hat f_0 + \hat g_0)$ is a randomized encoding of $f+g$ with output size $k + k' - |J| - 1$ and additive component $\hat f_0 + \hat g_0$.*

Remark 7. We can repeatedly apply Corollary 1 to find a randomized encoding of the concatenation of many functions. E.g. if we use the randomized encoding of our monomial $x_0\cdots x_{m-1}$ we get from Lemma 3 (beyond others) the components $f' = (x_{S_{1,i}} - a_i, (x_{S_{1,i}} a_{i+1,\ldots,j} - a_{i,\ldots,j})_{j\in\mathbb{Z}_{r_1}: j\neq i,i-1})$. If we now apply Lemma 3 to the first component and Lemma 4 (with some fixed $\mu \in \mathbb{Z}_{|I_{1,i}|}$) to all other components, then Corollary 1 (applied $(r_1 - 2)$ times) leads to a randomized

[12] Using encodings in different reconstructions is in general not secure (at least not for the straightforward combination of the simulators)—see [42]. Example 3.

encoding of the concatenation f' of output size $|I_{1,i}|^2 - |I_{1,i}| + 1 + (r_1 - 2)(|T_\mu| - |I_{1,i}| + 1)$. Please also note that exactly as in the special case, we see that the terms (i) in Lemma 4 are always already constructed by the corresponding Lemma 3 randomized encoding linear in the same terms; analogously for Lemma 5.

Remark 8. We can also use Corollary 1 to find a randomized encoding for additive terms like f_{add}. For example, if we take a randomized encoding of our monomial $x_0 \cdots x_{m-1}$ using Lemma 3 we get the additive component $f_{\mathsf{add}} = \sum_{i \in \mathbb{Z}_{r_1}} x_{S_{1,i}} a_{i+1,\ldots,i-1} - a_{\mathsf{add}}$. We assume that Lemma 3 has already been applied to the $x_{S_{1,i}}$, e.g. as part of the randomized encoding of f_{ii} and we are only interested, how many new outputs are needed to also construct f_{add}. Thus, if we apply Lemma 4 to each summand $x_{S_{1,i}} a_{i+1,\ldots,i-1}$ (where we consider once $x_{S_{1,0}} a_{1,\ldots,-1} - a_{\mathsf{add}}$ to account for the final constant), then we only need to construct the terms from (ii),(iii),(iv), since we assumed that (i) is already accounted for. Overall these are $r_1(|T_\mu| - r_1) + 1$ (additional) terms, where $r_1(|T_\mu| - r_1)$ comes from using Lemma 4 (ii), (iii) for each summand and the $+1$ comes from the sum of the r_1 additive (iv) terms that can be combined by Corollary 1 into one additive component.

Remark 9. Please also note that the previous Corollary 1 also applies to the randomness used in the additive components. Namely, if a_{add} is a summand of the additive component $\hat{f}_0$ and b_{add} is summand of the additive component $\hat{g}_0$, then $(a_{\mathsf{add}} + b_{\mathsf{add}})$ is obviously a summand of $\hat{f}_0 + \hat{g}_0$, although slightly more structured. In particular, even after applying the Corollary, we still have exactly one structured random number for each additive component and one unstructured random number for each other component of the overall randomized encoding.

In summary, we generate with our Lemma 3 a randomized encoding $\hat{f}^{(1)}$ of f linear in the $x_{S_{1,j}}$. We then generate for each component $\hat{f}^{(1)}_j$ of $\hat{f}^{(1)}$ a randomized encoding $\hat{f}^{(2)}_j$ linear in the $x_{S_{2,i}}$ using Lemmas 3 to 5 (and in the case of an additive term also Corollary 1). Corollary 1 allows us to concatenate the $\hat{f}^{(2)}_j$ into a randomized encoding $\hat{f}^{(2)}$ of $\hat{f}^{(1)}$. Finally the two encodings $\hat{f}^{(1)}$ and $\hat{f}^{(2)}$ can be composed with Lemma 2 to a randomized encoding of f linear in the $x_{S_{1,i}}$. We iterate over the previous steps until we arrive at a randomized encoding of f linear in the $x_{S_{\ell,j}}$. An algorithmic version of our construction is included in [42]. Protocol 2, where the output set contains the encodings of f linear in $x_{S_{\ell,j}}$. Please also consider [42]. Figure 8 which illustrates how the different encodings are combined under concatenation and composition.

4.4 Recursive Formula for Output Size

We next want to compute the output and randomness for *each* of our randomized encodings of $x_{S_{0,0}}$, i.e. for each choice of a series of refinements of partitions of $S_{0,0}$ or equivalently for each tree structure as in Fig. 3. Since our randomized encodings were constructed iteratively, we will also develop an iterative formula first. To this end, let $N^0_{S_{k,j}}$ be the number of level ℓ encodings linear in $x_{S_{\ell,i}}, 0 \leq$

$i < r_\ell$, needed to compute $x_{S_{k,j}} - c$ for some $c \in R$. Furthermore, let $N^1_{S_{k,j}}$ be the number of additional encodings needed to also construct $ax_{S_{k,j}} - c'$ for some $a, c' \in R$. Finally, let $N^2_{S_{k,j}}$ be the number of yet additional encodings needed to construct $abx_{S_{k,j}} - c''$ for some $a, b, c'' \in R$. Recall that these are just the cases (a), (b), (c) discussed in the special case above. From Lemma 3 we then get

$$\begin{aligned} N^0_{S_{k,j}} &= \sum_{i \in I_{k,j}} N^0_{S_{k+1,i}} + (|I_{k,j}| - 2) \sum_{i \in I_{k,j}} N^1_{S_{k-1,i}} + \sum_{i \in I_{k,j}} (N^1_{S_{k+1,i}} - 1) + 1 \\ &= \sum_{i \in I_{k,j}} N^0_{S_{k+1,i}} + (|I_{k,j}| - 1) \sum_{i \in I_{k,j}} N^1_{S_{k+1,i}} - |I_{k,j}| + 1 \end{aligned} \tag{2}$$

where the first sum corresponds to the $f^{(k+1)}_{ii}$. The factor $(|I_{k,j}| - 2)$ comes from choices of $j \neq i, i-1$ for each i in the $f^{(k+1)}_{ij}$. The third sum accounts for the additive term as in Corollary 1 and Remark 8, i.e. $\sum_{i \in I_{k,j}} (N^1_{S_{k+1,i}} - 1)$ for (ii),(iii) in Lemma 4 plus one additional additive term. We further get

$$N^1_{S_{k,j}} = \sum_{i \in I_{k,j} \setminus \{\mu\}} N^1_{S_{k+1,i}} |T'_\mu \cap M_i| + (|I_{k,j}| - 1) N^2_{S_{k+1,\mu}} + N^1_{S_{k+1,\mu}} - |I_{k,j}| + 1 \tag{3}$$

where $M_\iota = \{\iota\} \times I_{k,j}$ and the $T'_\mu := \psi_{kj}(T_{\psi^{-1}_{kj}(\mu)}), T'_\nu := \psi_{kj}(T_{\psi^{-1}_{kj}(\nu)})$ are defined as in Lemma 4 using the natural identifications $\psi_{kj} : \mathbb{Z}_{|I_{k,j}|} \to I_{k,j}$.[13] We receive this term again from Lemmas 3 to 5, where the additive term contributes $N^2_{S_{k+1,\mu}} + \sum_{i \in I_{k,j} \setminus \{\mu\}} N^1_{S_{k+1,i}} - |I_{k,j}| + 1$. The intersection $|T'_\mu \cap M_i \cap \{(i,j) : j \neq i-1\}| = |T'_\mu \cap M_i| - 1$ together with the sum over $i \neq \mu$ accounts for the cases (iii) in Lemma 4, the single $N^1_{S_{k+1,\mu}}$ for $g^{(k+1),a,\mu}_{\mu\mu}$. We also have the additional $(|I_{k,j} - 2|) N^2_{S_{k+1,\mu}}$ for the $g^{(k+1),a,\mu}_{\mu j}$ for $j \neq \mu, \mu - 1$ in (ii) of Lemma 4. Altogether we get Equation (3). Furthermore, we have

$$N^2_{S_{k,j}} = \sum_{i \in I_{k,j}} N^{1+|\{i\} \cap \{\mu,\nu\}|}_{S_{k+1,i}} |M_i \cap T'_\mu \cap T'_\nu| - |I_{k,j}| + 1 \tag{4}$$

The additive term is again constructed as before, where the $-|I_{k,j}| + 1$ results from using a sum over all (v) components as in Corollary 1. The summands of the additive term are combined as before with the cases $j \neq i - 1$ of (iii) and (iv) of Lemma 5. Similarly, the $h^{(k+1)}_{\mu\mu}, h^{(k+1)}_{\nu\nu}$ terms complement the exclusions $j \neq \mu$ in the other cases. Using $M_\mu \cap T'_\mu = M_\mu$ and $M_\nu \cap T'_\nu = M_\nu$ one can quickly deduce Eq. (4).

4.5 Application in MPC Protocols and Asymptotic Behavior

From Protocol 1 we already know how to use the new randomized encodings $\hat{f} = (y_l)_{0 \leq l < k}$ of $f(x_0, \ldots, x_{m-1}) = x_{S_{0,0}}$ in an MPC protocol. Following the

[13] While $N^1_{S_{1,i}}$ and $N^2_{S_{k,j}}$ below depend on μ and ν, these indices can be chosen freely, i.e. we can choose to which components we want to assign the prefactors. For this reason, we decided to not mark the two numbers with another μ or ν index.

discussion above, we know that the y_l consist of terms linear in $x_{S_{k,\ell}}$ for $j \in \mathbb{Z}_{r_\ell}$ and are of the form $f_*^{(l)}, g_*^{(l),*}, h_*^{(l)}$. Hence if we set $N^\gamma_{S_{\ell,j}} = 1$ for all $\gamma = 0, 1, 2$ (one for each y_l) Eqs. (2) to (4) allows us to compute the output size k. In our MPC Protocol 1 we have to send the resulting $k = N^0_{S_{0,0}}$ encodings plus the initial $|S_{0,0}| = m$ masked values $x_j - a_j$, i.e. we get bandwidth $N^0_{S_{0,0}} + m$.

We have seen in Sects. 4.2 and 4.3 that the y_l are multivariate polynomials in the input variables $x_0, \ldots, x_{m-1}$. They do not necessarily satisfy (I)–(III) in Sect. 4.1 yet. However, recall that we can rewrite multivariate polynomials like y_l in terms of the masked values $x_j - a_j$ as in (1) and then (I)–(III) are satisfied. The coefficients b_e of this expansion in the $x_j - a_j$ are binomial tuples, which are polynomials in the a_j and the randomized prefactors of y_l. In addition to the (structured) randomness in these binomial tuples, our construction also needs the randomness from the $a_{ij}, b_{ij}, c_{ij}, a_{\mathsf{add}}, b_{\mathsf{add}}, c_{\mathsf{add}}$ that result from Lemmas 3 to 5 and Corollary 1.[14] Hence we can define a *polytuple* as follows:

Definition 2. *Let f be a multivariate polynomial in $x_0, \ldots, x_{m-1}$ and $\hat{f}(x_0, \ldots, x_{m-1}, \tilde{a}_0, \ldots, \tilde{a}_{t'}) = (y_l)_{0 \le l < k}$ a randomized encoding of f constructed with our iterative approach, i.e. the $\tilde{a}_j$ are the $a_{i,\ldots,j}, b_{i,\ldots,j}, c_{i,\ldots,j}, a_{\mathsf{add}}, b_{\mathsf{add}}, c_{\mathsf{add}}$ which result from Lemmas 3 to 5 and Corollary 1. Then a* polytuple $[\![\hat{a}]\!]$ *to $\hat{f}$ consists of a shared structured random number $[\![\tilde{a}_j]\!]$ for each $\tilde{a}_j$, $0 \le j \le t'$, and one binomial tuple for each y_l, $0 \le l < k$.*

Remark 10. Recall from Sect. 3.4 that a term $x_S - a_S$ can be computed with a $2^{|S|} - 1$ binomial tuple for any finite set S; a term $ax_S - b_S$, as well as a term $abx_S + c_S$ for randomness $a, b, b_S, c_S \in R$, each need a binomial tuple of size $2^{|S|}$ compensating for the additional prefactor(s), i.e. in the notation of Sect. 3.4 a tuple $(ab_e)_{e \in E}$ or $(abb_e)_{e \in E}$.

Since we know from Lemmas 3 to 5 and the subsequent remarks that for each encoding we get exactly one new (possibly structured) random variable, we can also use the iterative formulas in Eqs. (2) to (4) to compute the polytuple size. Namely, if we replace $N^\gamma_{S_{k,j}}, \gamma = 0, 1, 2$, in Eqs. (2) to (4) by the corresponding tuple sizes $T^\gamma_{S_{k,j}}$ and set $T^0_{S_{\ell,j}} + 1 = T^1_{S_{\ell,j}} = T^2_{S_{\ell,j}} = 2^{|S_{\ell,j}|}$, then $T^\gamma_{S_{0,0}}$ will be the tuple size needed to compute $x_{S_{0,0}} = x_0 \cdots x_{m-1}$.

Please note that the size of a polytuple, as well as the output size of the randomized encoding strongly depend on the chosen tree structure (cf. Fig. 3), i.e. partitions. To better understand how the tree structure affects the asymptotic behavior of the bandwidth and tuple size, we consider trees with a fixed number $b = |I_{k,j}| \ge 1$ of factors multiplied in each node. Hence we can compute $x_{0,\ldots,m-1}$ for $m = \lambda b^n$ iteratively with $S_{n-k,j} = \{\lambda b^k \cdot j + i : 0 \le i < \lambda b^k\}, 0 \le j < b^{n-k}, 0 \le k \le n$, i.e. each degree b^k term splits into b encodings of degree b^{k-1} until we reach a level of elementary randomized encodings of degree $\lambda \ge 1$. For an explicit calculation in the special case $b = 2, n = 3, \lambda = 2$ we refer to [42].

[14] Recall from Sect. 3 that we also include terms deterministic in random variables in our randomness space.

Example 5. Now we can state the main result on the asymptotic behavior, which we prove in [42]. Appendix A.

Theorem 1. *Let $\lambda, b, S_{k,j}$ be defined as before. A product of $m = \lambda b^n$ shared inputs can be constructed with a polytuple of size $\mathcal{O}\left(2^\lambda\left(\frac{b^2+1}{2}\right)^n\right)$ with bandwidth $\mathcal{O}\left(\left(\frac{b^2+1}{2}\right)^n\right)$. In the special case $b = 2$, one only needs a tuple of size $2^{n-2}((2^\lambda - 1)n^2 + (2^{\lambda+2} - 2^\lambda + 1)n + 4(2^\lambda - 2)) + 1$. For $b = 2$, the bandwidth becomes $2^n n + 1 + m$.*

Remark 11. If we fix λ small, e.g. $\lambda \leq 3$, the case $b = 2$ leads to a bandwidth in $\mathcal{O}(m\log(m))$ and a tuple size in $\mathcal{O}(m\log(m)^2)$ while in all cases $b > 2$ both values are not even in $\mathcal{O}(m^2)$ (cf. Proof Theorem 1 and [42]. Lemma 6 in [42]. Appendix A). Furthermore, we remark that for a mixed number of factors going into a node the complexity will be dominated by the largest degree that occurs in a significant fraction of encodings. Finally, note that the complexity analysis also covers the case of a binomial tuple for $b = 1$.

Polynomials in Several Variables. Up to this point, we mainly discussed the computation of products $x_0 \cdots x_{m-1}$. However, the previous results directly transfer to general monomials $x^{\boldsymbol{d}} = x_0^{d_0} \cdots x_{m-1}^{d_{m-1}}, \boldsymbol{d} = (d_0, \dots, d_{m-1})$ simply by replacing the variables x_i in the randomized encoding by $x_i^{d_i}$. A component of the randomized encoding will then be linear in $\prod_{s \in S_{\ell,j}} x_s^{d_s}$ and can still be constructed using a binomial tuple. From Sect. 3.4 we know that $T^0_{S_{\ell,j}} = T^1_{S_{\ell,j}} - 1 = T^2_{S_{\ell,j}} - 1 = \prod_{s \in S_{\ell,j}}(d_s + 1) - 1$. For the special case where $|S_{\ell,j}| = 1$, e.g. $S_{\ell,j} = \{j\}$, we have $T^0_{\{j\}} = d_j + 1$, i.e. $[\![x_j^{d_j} - a'_{j,d_j}]\!] = -[\![a'_{j,d_j}]\!] + \sum_{i=0}^{d_j}[\![a_j^i]\!](x_j - a_j)^{d_j - i}$ for a new mask a'_{j,d_j}. Then the tuple size needed to compute $x_0^{d_0} \cdots x_{m-1}^{d_m}$ follows recursively from Eqs. (2) to (4). If $d_j = d/m \in \mathbb{N}$ the tuple size to compute $x_0^{d_0} \cdots x_{m-1}^{d_{m-1}}$ for $m = 2^n$ becomes $2^{n-2}((\frac{d}{m})n^2 + (3\frac{d}{m} + 4)n + 4\frac{d}{m} - 4) + 1$. For details we refer to the proof of Theorem 1 in [42]. Appendix A which contains the formulas (and proof thereof) whenever $T^1_{S_{\ell,j}} = T^2_{S_{\ell,j}}$. The result shows that in the total degree $d = \sum_{j=0}^{m-1} d_j$ we can get down to complexity $\mathcal{O}(d\log(m)^2)$ in the tuple size. The same bound on the complexity also holds for all other cases with $d = \sum_{j=0}^{m-1} d_j$ since we can choose μ, ν in Eqs. (3) and (4) always such that the encodings with randomized coefficients are linear in those $x_{S_{\ell,j}}$ for which $T^1_{S_{\ell,j}} = T^2_{S_{\ell,j}}$ is minimal, i.e. from the cases with $d_j \leq d/m$. Please recall that $x_0^{d_0} \cdots x_{m-1}^{d_{m-1}}$ was already discussed in the introduction in Table 1.

Finally, we can combine the randomized encodings (and corresponding polytuples) for different monomials in a general polynomial f with Corollary 1. Namely, if we have two randomized encodings f, g (as constructed before), we need to generate common components only once and we can add the components corresponding to $\hat{f}_0$ and $\hat{g}_0$ in Corollary 1. Observe that all our encodings have the specific form expected by Corollary 1, i.e. have an additive component.

Overall we find for any multivariate polynomial f a randomized encoding and a corresponding polytuple.

4.6 Composability and Security

From Sect. 4.2 we know how to evaluate a polynomial $f(x_0, \dots, x_{m-1})$ in a single round using polytuples. With our MPC protocol $\Pi_{\text{polynomial}}$ presented in Sect. 4.1 (cf. also [42]. Protocol 5), we are able to do this in three different ways: (i) compute $f(x_0, \dots, x_{m-1})$ publicly (i.e. the result is an output of the function to be evaluated with MPC), (ii) compute $[\![f(x_0, \dots, x_{m-1})]\!]$ (this can be used in other subprotocols that require their inputs as shares), and (iii) compute $f(x_0, \dots, x_{m-1}) - b$ where b is part of the tuple for another polynomial g; this allows our protocol to be used in a multi-round fashion. While (i) and (ii) are straightforward applications of the results from the previous subsections, we want to take a closer look at the multi-round use, which allows a different form of tradeoff. Namely, we allow a (slightly) larger number of communication rounds but can therefore further reduce the tuple size and bandwidth.

Multi-round Evaluation. Assume the parties have agreed on a series of polynomials $f_j, 0 \le j < m$ with input tuples X_j (not-necessarily disjoint) and a polynomial f in m variables. They want to compute $f(f_0(X_0), \dots, f_{m-1}(X_{m-1}))$. Thus, they agree on one of our randomized encodings for each f_j and f. The parties construct the corresponding polytuples $[\![A_j]\!], 0 \le j < m$ (for each f_j) and $[\![A]\!]$ (for f) in the preprocessing phase and receive inputs $[\![X_j]\!]$ in the input phase. They run $\Pi_{\text{polynomial}}(X_j, f_j, \mathsf{continuation} := (f, j))$ in parallel to receive $(x_\iota - a_\iota), 0 \le \iota < |X_j|, 0 \le j < m$ in a single broadcast round. Then the parties locally compute the shares of the elementary encodings and adjust an additive component by $[\![a_j]\!]$ such that after the next broadcast every party can locally compute the public values $z_j := f_j(X_j) - a_j$. Finally, they call $\Pi_{\text{polynomial}}((z_1, \dots, z_m), f, \mathsf{continuation} := \text{open})$. Observe that in this call, the first step of $\Pi_{\text{polynomial}}$ does not require any opening of elements as all z_j are already public masked values.

Remark 12. Our protocol is also compatible with techniques used in Turbospeedz [7] and ABY2.0 [41] that use function-dependent preprocessing. This allows to reduce the online bandwidth even more. As an extreme case, one would only have to open the randomized encoding without the $x_j - a_j$ which are then already accounted for. Using only Beaver multiplication (or binomial tuples), this would exactly correspond to the complexity of ABY2.0 or Turbospeedz.

In Sects. 4.2 to 4.5 we have seen that by suitably choosing the randomized encodings and corresponding polytuples, we can trade-off bandwidth and tuple size while keeping the round complexity minimal. The multi-round feature adds additional flexibility to our online phase. In particular, it allows us to increase the round complexity slightly to prevent possible performance bottlenecks in bandwidth and tuple size. Figure 1 illustrates this tradeoff between round complexity, bandwidth, and tuple size. Please also see [42]. Example 1 in [42]. Appendix A for an explicit example. We remark that once the polynomial to be evaluated and the network setup are known, a compiler can use the exact calculations of tuple size and bandwidth from Equation (2) to determine the best performing polytuple solution before the actual computation starts. Furthermore, ideal solutions for classical and regularly used setups can be hard-coded.

Security. Our protocol $\Pi_{\text{polynomial}}$ and the resulting full online protocol[15] Π_{online} (cf. [42]. Protocol 5) are secure and composable in the sense of *universal composability* (UC) [12], i.e. they can be combined with other MPC protocols, while still giving the same guarantees as an idealized protocol (a so-called functionality). For the corresponding ideal functionalities see [42]. Appendix B.

Let $[\![X]\!]$ be a tuple of authenticated inputs to a polynomial f and $[\![A]\!]$ the respective tuple. Intuitively, the security of our approach can be argued as follows: All opened values apart from one additive component of the randomized encoding are masked with a new random element from $[\![A]\!]$, i.e. they are encrypted with a one-time pad and hence are information-theoretically secure. The final additive encoding contains the result minus a public constant (constructed from the other (pseudo)random components of the randomized encoding). In particular, it contains no more information than the result itself.

All values that are opened are authenticated and thus their integrity can be checked with the usual aggregated MAC check (cf. [42]. Protocol 7; recall that we now consider R to be a finite field). In particular, our MAC check Π_{CheckMAC} is chosen identical to the classical MAC-check in [21]. Formally, we then have the following security result for the online protocol Π_{online} in [42]. Protocol 5:

Theorem 2. *The protocol Π_{online} realizes $\mathcal{F}_{\text{online}}$ in the $(\mathcal{F}_{[\![\cdot]\!]}, \mathcal{F}_{\text{random}}, \mathcal{F}_{\text{commit}})$-hybrid model with statistical security against any active adversary corrupting up to $n-1$ parties.*

Proof. The proof of this theorem is mostly the same as the security proofs for the corresponding online protocols in [21,22]. Both construct a suitable simulator, e.g. [21, Fig. 22]. The only difference for a simulator in our protocol is in polynomial operations that are opened (i.e. calls to $\Pi_{\text{polynomial}}$ with $\mathsf{continuation} = \text{open}$). Recall that the simulator works on *random* inputs (instead of the real inputs for honest (input) parties) and simulates the protocol run with these inputs. It will then receive an output z of the simulation that is most likely wrong. However, the ideal functionality $\mathcal{F}_{\text{online}}$ provides the simulator with the real output y. The simulator adjusts the share of the additive encoding[16] y_0 of one (simulated) honest party P_i by $\Delta = y - z$, i.e. $[y_0]_i \to [y_0]_i + \Delta$. Since the simulator also knows the MAC key α, it can change $[\alpha y_0]_i \to [\alpha y_0]_i + \alpha\Delta$. Thus the MAC check for the result will pass (if corrupted parties did not misbehave) and the result will be the same in the real and ideal world. □

[15] Recall that apart from the $\Pi_{\text{polynomial}}$ subprotocol, our online protocol Π_{online} coincides with the online protocols from other SPDZ-like protocols like [21,31].

[16] Recall that our construction always comes with an additive encoding.

4.7 The Generation of Polytuples

In the previous paragraphs, we have seen how to build an actively secure MPC online [42]. Protocol 5 which consumes polytuples. Of course, the polytuples have to be generated first in an offline phase, which can run well before the actual input data (the x_j) becomes available. Since polytuples are entry-wise just multivariate polynomials in random numbers, the parties can invoke any MPC protocol that can provide (authenticated) shares of such terms. For example, for an actively secure offline phase we can plug in any of the protocols [22,30,31,44] to first generate a sufficient number of Beaver triples. The parties can then use these Beaver triples to multiply shared random numbers, e.g. they run the standard online protocol within the offline phase on the random numbers (instead of actual inputs). Hence they can construct each entry of the polytuple.

The number of Beaver triples needed for this approach can again be computed by an iterative formula. The result are the Eqs. (2) to (4) each shifted $+|I_{k,j}|$. Recall from Corollary 1 that we did combine all additive terms into one constant and hence reduced the output and tuple size by $-|I_{k,j}|$. At the same time, the new additive term became more complex, namely a sum of the original monomials in the separate additive components. Even after combining the additive terms, we still need to build each of these monomials with Beaver triples. Thus the reduction of output and tuple size does not carry over to this generic offline approach and we have to add $|I_{k,j}|$ in the iterative formulas.

Exactly as in the proof of Theorem 1 we can then deduce that the number of Beaver triples needed (in the case $b = 2$ of binary trees) is still in $\mathcal{O}(m \log(d)^2)$ but with slightly larger constant. For example, in the case $b = 2$ we then need $2^{n-2}((\frac{d}{m} + 1)n^2 + (3\frac{d}{m} - 1)n + 4\frac{d}{m}) - 1$ Beaver triples if we use $d_i - 1$ Beaver triples to compute $\left[a^{d_1}\right]$ from $[a]$—of course, this is a rough estimate given that we often can compute the power with around $\log(d_1)$ Beaver triples (cf. also [42]. Remark 14 in [42]. Appendix A).

To simply plugin established offline protocols comes with certain advantages, e.g. that implementations already exist and that we can profit from their future optimizations. However, this approach is not optimized for the use with polytuples. In [42]. Appendices C and D we therefore present different new solutions for an actively secure tuple generation (e.g. an extended sacrificing [42]. Protocol 10).

Finally, please recall that our approach is not restricted to the case of binary trees or 1(+1) round protocols. In particular, if the generation of $\mathcal{O}(m \log(d)^2)$ Beaver triples is too slow, the parties can use a different number of rounds and different randomized encodings to get an ideal performance for their use case.

5 Implementation and Evaluation

To illustrate the practicality of our approach, we implemented the online phase in the MP-SPDZ framework [29] and ran several benchmarks. Furthermore, we implemented the plugin offline phase from Sect. 4.7 which uses Beaver triples to generate the polytuples. Our implementations are available at [43]. These first benchmarks show that we can outperform the standard Beaver triple-based approach in the online phase for all tested applications. Our benchmarks include (i) evaluation of multivariate polynomials, (ii) establishing a ranking of inputs (e.g. for auctions or e-voting), and (iii) evaluating neural networks. We ran the experiments on a single machine (laptop with an i7-8565U CPU, 1.80 GHz) where each party runs on a single core/thread. We simulated different network settings for $n = 2$ parties with standard Linux tools (see [42]. Appendix G for details). All tested latency settings are rather conservative and roughly correspond to parties located in the same country or continent. The tested latencies are significantly lower than the 40 ms assumed in the WAN setting (e.g. in [40]). The trends in all benchmarks show that our approach will perform even better in such a setting.

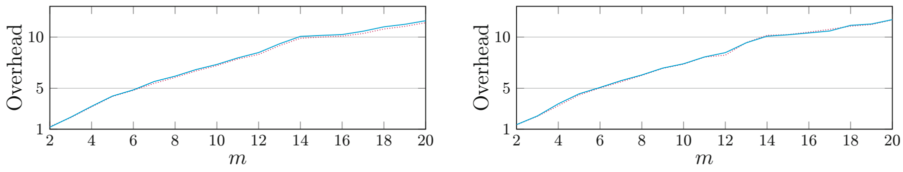


Fig. 4. The left diagram shows the bandwidth overhead of the polytuple plugin offline phase compared to classical SPDZ-like protocols for the computation of $x_0 \cdots x_{m-1}$. The right diagram shows the corresponding runtime overhead. For the blue line we used the LowGear offline protocol [31], for the red dotted line the MASCOT protocol [30].

With our implementation, we added elementary operations for powers and products to MP-SPDZ. We use polytuples of minimal tuple size as in Theorem 1 for $b = 2$. Furthermore, we implemented the case $b = m$, i.e. the case where polytuples become binomial tuples. For both variants, we also implemented a prefix variant (along [42]. Appendix E) used for comparison in our benchmarks below. Moreover, our implementation supports MP-SPDZ's parallelism model: arbitrarily many operations of the same type can be combined and executed in one step (reducing the number of communication rounds).

Next, we describe our test applications and discuss the results of our benchmarks. We always compare our implementation for $b = 2$ against the state-of-the-art implementation from MP-SPDZ. We do not compare to the binomial tuples case since first benchmarks showed that the local computation times for the tuple production are beyond practical (as expected by the large tuple size).

Polynomial Evaluation. As an example of a polynomial evaluation, we chose the power series expansion of a multivariate Gauss functions $\exp(-\langle x, x\rangle/2)$ up to degree d in each variable. This polynomial is then simply evaluated by computing all needed (prefix) powers of all variables and multiplying them with our polytuples. We compare this to the same computation with standard (Beaver

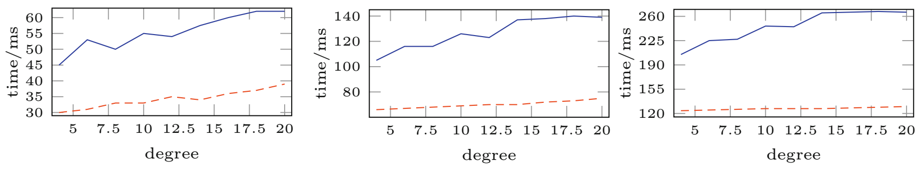
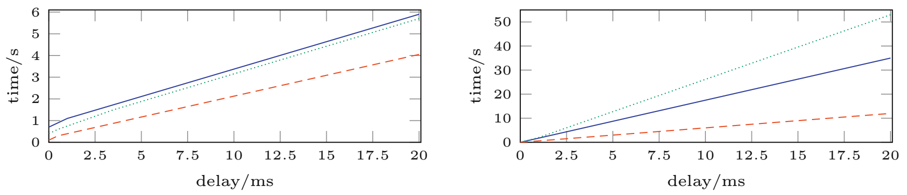


Fig. 5. Benchmarks for Gaussian with 32 variables with 2 ms (left), 5 ms (middle), 10 ms (right) delay; (blue: default MP-SPDZ implementation, orange/dashed: ours). (Color figure online)

triple-based) tools included in MP-SPDZ. Figure 5 and [42]. Figure 9 show the results for this benchmark. Our approach has a clear advantage in runtime—even for very small network delays of only 2 ms. Note that also the bandwidth is lower with our approach. For the Beaver-based implementation, we can clearly see the effect of a logarithmic number of rounds on the runtime, while our approach has an almost constant runtime (in the degree of the polynomial).

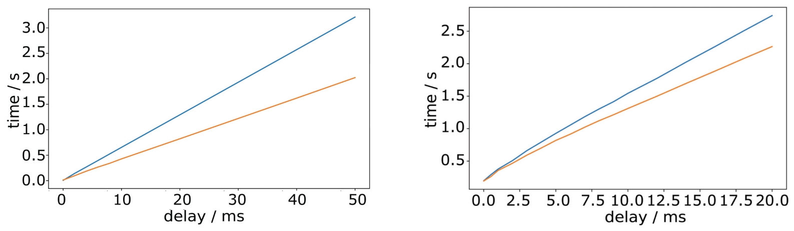
Rankings. For auctions (or e-voting), one often needs to compute a ranking of the bids (or votes) and reveal the top k results (e.g. with $k = 1$ only the highest bid or the candidate with the most votes). There are several established methods to compute these rankings. For our evaluation, we chose two approaches, one purely based on inequality tests and one which uses equality and inequality tests. In order to use our new protocols to speed up the comparison we use bitwise comparisons as in [20] which allow us to employ polytuples. For details, we refer to [42]. Appendices E and F. We benchmarked both approaches with our polytuples-based protocol and compare them to the respective default implementation in MP-SPDZ (based on the protocols with logarithmic complexity in [13]; with and without edabits [23] to speed up the comparison). We compute rankings of $m = 40$ items (bids or candidates). The benchmark results in Fig. 6 show that our new approach is faster than the others.



(a) Using pairwise inequality tests.

(b) Using inequality and equality tests.

Fig. 6. Benchmarks for rankings (blue: default MP-SPDZ implementation, orange/dashed: ours, green/dotted: MP-SPDZ with edabits [23] (Color figure online)).



(a) ArgMax Layer, unlimited rate. **(b)** Network A [39].

Fig. 7. Benchmarks for an ArgMax layer and the evaluation of a sample neural network included in MP-SPDZ [29] as network A (cf. [45]; blue: default MP-SPDZ, orange: ours) both without bandwidth restriction. For further benchmarks see [42]. Figure 10.

Remark 13. SPDZ is a protocol originally designed for an arithmetic circuit evaluation and not for comparisons. In particular, there other MPC approaches better suited for some types of comparisons. However, our goal is to extend SPDZ and hence in particular to avoid expensive conversations to some other scheme. We therefore decided to compare our evaluation for comparisons also to SPDZ, although there are other competitive MPC protocols.

Neural Networks. Among others, MP-SPDZ [29] contains examples of deep neural networks. For our benchmarks, we ran the networks labeled A [39], B [37], C [34], and D [45] (as in [32,47]). Each of these networks has a final ArgMax layer (see [42]. Appendix F for the specific layers). Replacing *only this single layer* with a polytuple-based comparison (see [42]. Appendix F for details) can already have a noticeable impact on the overall runtime of the network, as can be seen in Fig. 7. We also remark that a bandwidth rate restriction does not affect the performance and hence the theoretical bandwidth overhead of the polytuples approach is negligible in our example (see e.g. [42]. Figure 10 in [42]. Appendix G).

Tuple Generation. Finally, we also benchmarked the offline phase for the plugin approach described in Sect. 4.7. Our first results in Fig. 4 confirm our theoretical results of Sect. 4, i.e. we get a log-linear overhead over SPDZ independent of the employed offline protocol (Overdrive LowGear [31] and MASCOT [30]). As our focus is on applications where the offline phase is not time-critical, we leave further benchmarking of the offline phase and possibly improving the polytuple generation (e.g. as in [42]. Appendix C) to future work.

Overall, our evaluation shows that our approach has a clear performance advantage over SPDZ in the online phase for classical sample applications like the evaluation of multivariate polynomials or comparisons.

Acknowledgments. This research was supported by the CRYPTECS project founded by the German Federal Ministry of Education and Research under Grant Agreement No. 16KIS1441 and by the French National Research Agency under Grant Agreement No. ANR-20-CYAL-0006 and by Advantest as part of the Graduate School "Intelligent Methods for Test and Reliability" (GS-IMTR) at the University of Stuttgart. Additionally, this research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Fundation) - 411720488. Furthermore, Toomas Krips was partly supported by the Estonian Research Council, ETAG, through grant PRG 946. We furthermore thank Simon Egger for his help and valuable remarks.

References

1. Applebaum, B., Brakerski, Z., Tsabary, R.: Perfect secure computation in two rounds. In: Theory of Cryptography. pp. 152–174. Springer (2018)
2. Applebaum, B., Brakerski, Z., Tsabary, R.: Degree 2 is complete for the round-complexity of malicious mpc. In: EUROCRYPT 2019. pp. 504–531. Springer (2019)
3. Applebaum, B., Ishai, Y., Kushilevitz, E.: Cryptography in NC$^\circ$. SIAM Journal on Computing **36**(4), 845–888 (2006)
4. Bar-Ilan, J., Beaver, D.: Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In: PODC 1989. pp. 201–209. ACM (1989)
5. Baum, C., Cozzo, D., Smart, N.P.: Using TopGear in Overdrive: A more efficient ZKPoK for SPDZ. In: SAC 2019. pp. 274–302. Springer (2020)
6. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: CRYPTO '91. pp. 420–432. Springer (1992)
7. Ben-Efraim, A., Nielsen, M., Omri, E.: Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing. In: ACNS 2019. pp. 530–549. Springer (2019)
8. Bitan, D., Dolev, S.: Optimal-Round Preprocessing-MPC via Polynomial Representation and Distributed Random Matrix (extended abstract). IACR Cryptol. ePrint Arch. **2019**, 1024 (2019)
9. Boura, C., Chillotti, I., Gama, N., Jetchev, D., Peceny, S., Petric, A.: High-precision privacy-preserving real-valued function evaluation. In: FC 2018. pp. 183–202. Springer (2018)
10. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient Pseudorandom Correlation Generators: Silent OT Extension and More. In: CRYPTO 2019. pp. 489–518. Springer (2019)
11. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In: ITCS 2012. pp. 309–325. ACM (2012)
12. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In: FOCS 2001. pp. 136–145. IEEE Computer Society (2001)
13. Catrina, O., de Hoogh, S.: Improved Primitives for Secure Multiparty Integer Computation. In: SCN 2010. pp. 182–199. Springer (2010)
14. Chen, H., Kim, M., Razenshteyn, I.P., Rotaru, D., Song, Y., Wagh, S.: Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning. In: ASIACRYPT 2020. pp. 31–59. Springer (2020)
15. Cho, H., Wu, D., Berger, B.: Secure genome-wide association analysis using multiparty computation, supplementary notes 3. Nat. Biotechnol. **36**(6), 547–551 (2018)
16. Couteau, G.: A note on the communication complexity of multiparty computation in the correlated randomness model. In: EUROCRYPT. pp. 473–503. Springer (2019)

17. Cramer, R., Damgård, I.: Secure Distributed Linear Algebra in a Constant Number of Rounds. In: CRYPTO 2001. pp. 119–136. Springer (2001)
18. Cramer, R., Fehr, S., Ishai, Y., Kushilevitz, E.: Efficient multi-party computation over rings. In: Biham, E. (ed.) Advances in Cryptology — EUROCRYPT 2003. pp. 596–613. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
19. Dahl, M.: Cryptography and machine learning (2017), Blog on the SPDZ protocol - part 2
20. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In: TCC 2006. pp. 285–304. Springer (2006)
21. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority – or: Breaking the SPDZ limits. In: ESORICS 2013. pp. 1–18. Springer (2013)
22. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO. pp. 643–662. Springer (2012)
23. Escudero, D., Ghosh, S., Keller, M., Rachuri, R., Scholl, P.: Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In: CRYPTO 2020. pp. 823–852. Springer (2020)
24. Ghodosi, H., Pieprzyk, J., Steinfeld, R.: Multi-party computation with conversion of secret sharing. Des. Codes Cryptogr. **62**(3), 259–272 (2012)
25. Hasler, S., Reisert, P., Rivinius, M., Küsters, R.: Multipars: Reduced-Communication MPC over Z2k. Proceedings on Privacy Enhancing Technologies (2), 5–28 (2024)
26. Ishai, Y., Kushilevitz, E.: Randomizing polynomials: A new representation with applications to round-efficient secure computation. In: FOCS. pp. 294–304 (2000)
27. Ishai, Y.: Randomization techniques for secure computation. In: Secure Multi-Party Computation (2013)
28. Ishai, Y., Kushilevitz, E., Meldgaard, S., Orlandi, C., Paskin-Cherniavsky, A.: On the Power of Correlated Randomness in Secure Computation. In: TCC 2013. pp. 600–620. Springer (2013)
29. Keller, M.: MP-SPDZ: A Versatile Framework for Multi-Party Computation. In: CCS '20. pp. 1575–1590. ACM (2020)
30. Keller, M., Orsini, E., Scholl, P.: MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In: CCS 2016. pp. 830–842. ACM (2016)
31. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In: EUROCRYPT 2018. pp. 158–189. Springer (2018)
32. Keller, M., Sun, K.: Secure Quantized Training for Deep Learning. CoRR **abs/2107.00501** (2021)
33. Kolesnikov, V.: Gate evaluation secret sharing and secure one-round two-party computation. In: ASIACRYPT 2005. pp. 136–155. Springer (2005)
34. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
35. Lin, H., Liu, T.: Two-Round MPC Without Round Collapsing Revisited – Towards Efficient Malicious Protocols. In: CRYPTO. pp. 353–382. Springer (2022)
36. Lin, H., Liu, T., Wee, H.: Information-theoretic 2-round MPC without round collapsing: adaptive security, and more. In: TCC 2020. pp. 502–531. Springer (2020)
37. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious Neural Network Predictions via MiniONN Transformations. In: CCS 2017. pp. 619–631. ACM (2017)
38. Lu, D., Yu, A., Kate, A., Maji, H.K.: Polymath: Low-Latency MPC via Secure Polynomial Evaluations and Its Applications. PETS 2022 (1), 396–416 (2022)

39. Mohassel, P., Zhang, Y.: SecureML: A System for Scalable Privacy-Preserving Machine Learning. In: SP 2017. pp. 19–38. IEEE Computer Society (2017)
40. Ohata, S., Nuida, K.: Communication-Efficient (Client-Aided) Secure Two-Party Protocols and Its Application. In: FC 2020. pp. 369–385. Springer (2020)
41. Patra, A., Schneider, T., Suresh, A., Yalame, H.: ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In: USENIX Security 2021. pp. 2165–2182. USENIX Association (2021)
42. Reisert, P., Rivinius, M., Krips, T., Hasler, S., Küsters, R.: Actively Secure Polynomial Evaluation from Shared Polynomial Encodings (Full Version). Crypt. ePrint 2024/1435 (2024)
43. Reisert, P., Rivinius, M., Krips, T., Hasler, S., Küsters, R.: Implementation to *Actively Secure Polynomial Evaluation from Shared Polynomial Encodings* (2024), Website of the Insitute of Information Security Stuttgart
44. Reisert, P., Rivinius, M., Krips, T., Küsters, R.: Overdrive LowGear 2.0: Reduced-Bandwidth MPC without Sacrifice. In: ACM ASIA CCS 2023 (2023)
45. Riazi, M.S., Weinert, C., Tkachenko, O., Songhori, E.M., Schneider, T., Koushanfar, F.: Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In: AsiaCCS 2018. pp. 707–721. ACM (2018)
46. Rivinius, M., Reisert, P., Hasler, S., Küsters, R.: Convolutions in Overdrive: Maliciously Secure Convolutions for MPC. In: PETS 2023 (2023)
47. Wagh, S., Gupta, D., Chandran, N.: SecureNN: 3-Party Secure Computation for Neural Network Training. Proc. Priv. Enhancing Technol. **2019**(3), 26–49 (2019)

Efficient Fuzzy Private Set Intersection from Fuzzy Mapping

Ying Gao[1,2](✉), Lin Qi[1], Xiang Liu[1], Yuanchao Luo[1], and Longxin Wang[1]

[1] School of Cyber Science and Technology, Beihang University, Beijing, China
{19373287,lx1234,19373507,wlx_buaa,gaoying}@buaa.edu.cn
[2] Zhongguancun Laboratory, Beijing, China

Abstract. Private set intersection (PSI) allows Sender holding a set X and Receiver holding a set Y to compute only the intersection $X \cap Y$ for Receiver. We focus on a variant of PSI, called fuzzy PSI (FPSI), where Receiver only gets points in X that are at a distance not greater than a threshold from some points in Y.

Most current FPSI approaches first pick out pairs of points that are potentially close and then determine whether the distance of each selected pair is indeed small enough to yield FPSI result. Their complexity bottlenecks stem from the excessive number of point pairs selected by the first picking process. Regarding this process, we consider a more general notion, called fuzzy mapping (Fmap), which can map each point of two parties to a set of identifiers, with closely located points having a same identifier, which forms the selected point pairs.

We initiate the formal study on Fmap and show novel Fmap instances for Hamming and L_∞ distances to reduce the number of selected pairs. We demonstrate the powerful capability of Fmap with some superior properties in constructing FPSI variants and provide a generic construction from Fmap to FPSI.

Our new Fmap instances lead to the fastest semi-honest secure FPSI protocols in high-dimensional space to date, for both Hamming and general $L_{\mathsf{p}\in[1,\infty]}$ distances. For Hamming distance, our protocol is the first one that achieves strict linear complexity with input sizes. For $L_{\mathsf{p}\in[1,\infty]}$ distance, our protocol is the first one that achieves linear complexity with input sizes, dimension, and threshold.

Keywords: Fuzzy private set intersection · Fuzzy mapping · Multi-query fuzzy reverse private membership test

1 Introduction

Private set intersection (PSI) enables two parties, each with a private set, to compute the intersection of their sets without revealing any information more than the intersection itself. Since its high practical value in threat detection, private contact discovery, sample alignment, and other scenarios, numerous PSI protocols [17,21,22] have been designed in last decades. And recent PSI protocols have achieved extremely high efficiency [21]. Facing various complex practical needs, there is also a growing interest in works on variants of PSI, including labeled PSI (LPSI) [3,5,9], which outputs labels associated with elements in intersection to Receiver; PSI cardinality (PSI-card) [10,16,25], which only reveals intersection cardinality to Receiver.

This work focus on a variant of PSI, fuzzy PSI (FPSI). The input of FPSI consists of m points from Sender in a d-dimensional space and n points from Receiver in the same space. And FPSI's output only informs Receiver those Sender's points that have the distance (e.g., Hamming distance, L_2 distance, etc.) with some Receiver's points not greater than the threshold δ, while nothing is revealed to Sender. FPSI has many potential applications in fields that involve fuzzy matching on datasets, such as privacy-preserving biometric search [23], illegal content detection [1], and vulnerable password detection [4]. For instance, the deployment of biometric systems in public places for searching for sensitive groups (such as fugitives) yields significant benefits to public safety. However, public concerns over privacy protection make it impractical to upload locally recognized biometric features to a cloud database for matching. Using PSI can solve the privacy issue, but since biometric features always contain some noises (e.g., due to environmental disturbance, algorithms' randomness, etc.), conventional PSI cannot fulfill feature matching. In such cases, FPSI becomes indispensable.

Since the concept of fuzzy matching was introduced in PSI by Freedman, Nissim, and Pinkas [11] in 2004, there has been a long list of works related to FPSI [1,4,7,8,13–15,23,26]. The majority of them concern with FPSI for Hamming distance, and the few exceptions [13–15] only consider about one or two of L_1, L_2, and L_∞ distances. Until 2024, Baarsen and Pu [1] make a breakthrough by presenting the first FPSI protocol supporting general L_{p} distance with $\mathsf{p} \in [1, \infty]$. As the state-of-the-art FPSI for $L_{\mathsf{p}\in[1,\infty]}$ distance, the communication and computation costs of their protocols for high dimension scale linearly or quadratically with the dimension d. They also conduct research on some variants of FPSI, including labeled FPSI (LFPSI), fuzzy PSI-card (FPSI-card), and FPSI with sender privacy (FPSI-SP). Regrettably, due to the super-linear factor in complexities, their protocols still have room for improvement.

1.1 Motivation

Current FPSI protocols might have expensive overheads for their super-linear factors in complexities.

First, all existing FPSI protocols for Hamming distance retain super-linear factors with input sizes in their complexities. Starting from [11], most FPSI protocols for Hamming distance employ the same idea: perform fuzzy matching over all $m \cdot n$ pairs of inputs in order to select the final result. Existing works often focus on improving fuzzy matching protocol for Hamming distance, and rarely deal with $m \cdot n$ factor introduced by this idea. The current best FPSI protocol for Hamming distance reducing this quadratic factor at the cost of introducing assumption on inputs from both parties, only achieves near-linear complexity [8].

Second, the efficiency of existing FPSI protocols for $L_{\mathsf{p}\in[1,\infty]}$ distance is also not satisfactory. Using oblivious key-value store (OKVS) and decisional Diffie-Hellman (DDH) tuple, Baarsen and Pu [1] provide FPSI protocols with previous optimal complexities. However, most of their protocols are still troubled by super-linear complexity with dimension. Although their protocol based on locality-sensitive hashing (LSH) has communication and computation costs scaling linearly with dimension, its costs scale super-linearly with Receiver's input size. Unfortunately, the prevalence of real databases with substantial dimension and size (such as facial feature databases) makes the previously mentioned flaws greatly hindering the applications of their protocols.

So, there exist two fascinating open questions:

- *Can we construct an FPSI protocol for Hamming distance with communication and computation complexities that are strictly linear with m and n*[1]*?*
- *Can we construct an FPSI protocol for $L_{\mathsf{p}\in[1,\infty]}$ distance of which costs scale linearly with anyone of m, n, d, and δ?*

1.2 Our Contribution

We provide affirmative answers to these two questions in the semi-honest setting. Our main contributions are summarized as below.

- **A New Cryptographic Primitive Called Fuzzy Mapping.** We introduce the abstraction of a new cryptographic primitive called fuzzy mapping (Fmap). We show that many FPSI protocols [1,4,7,11,13,15,23,26] actually are based on instances of Fmap, and complexity bottlenecks in these protocols are derived from the excessive expansion rates of their Fmap instances. Under some reasonable assumptions about inputs, we present a non-trivial Fmap instance for Hamming distance and an Fmap instance for L_∞ distance with expansion rate of 1.
- **FPSI for Hamming Distance of Which Costs Scale Strictly Linearly with $\boldsymbol{m}$ and $\boldsymbol{n}$.** We provide a generic construction for FPSI from Fmap that does not introduce any additional assumptions about inputs. As an instance of it, we construct an FPSI protocol for Hamming distance using our Fmap instance for Hamming distance. Due to the employment of this non-trivial Fmap instance, communication and computation complexity of the new protocol achieve strict linearity with m and n for the first time.
- **FPSI for $\boldsymbol{L_{\mathsf{p}\in[1,\infty]}}$ Distance of Which Costs Scale Linearly with Anyone of $\boldsymbol{m}$, $\boldsymbol{n}$, $\boldsymbol{d}$, and $\boldsymbol{\delta}$.** We show how to construct multi-query fuzzy reverse private membership test (mqFRPMT), the fuzzy version of multi-query reverse private membership test (mqRPMT), from Fmap without expansion on Sender's set. Using mqFRPMT, we can easily obtain FPSI and

[1] That is to say, as both m and n grow to be k times larger, communication and computation costs of the protocol increase to k times at most.

its variants, including FPSI-card, LFPSI, and FPSI-SP. By instantiating with our Fmap instance for L_∞ distance with expansion rate of 1, we ultimately construct a new FPSI protocol for $L_{\mathsf{p}\in[1,\infty]}$ distance. Its costs scale linearly with any one of m, n, d, and δ, which allows it to perform better than prior protocols.

- **Performance.** Our experimental results demonstrate that compared with the state-of-the-art protocols, our protocol achieves a $4.6\times$ reduction in communication cost for Hamming distance when both parties input 128-bit binary strings and δ is set to 4, and achieves a $28-166\times$ speedup and $6-40\times$ reduction in communication cost for $L_{\mathsf{p}\in\{1,2,\infty\}}$ distance when $d \geq 6$.

1.3 Related Work

We review previous semi-honest secure FPSI protocols, which can be divided into two categories: FPSI for Hamming distance and FPSI for $L_{\mathsf{p}\in[1,\infty]}$ distance. A comparison of asymptotic complexities is given in Table 1.

FPSI for Hamming Distance. Freedman et al. [11] first propose the concept of FPSI and provide a protocol for Hamming distance based on polynomial interpolation and additively homomorphic encryption (AHE). Their protocol has been proved insecure by Chmielewski and Hoepman [7]. For a long time, subsequent works on FPSI mainly focus on Hamming distance. Ye et al. [26] design FPSI for Hamming distance with oblivious polynomial evaluation technique. Indyk and Woodruff [15] deal with FPSI for Hamming and L_2 distances, but their protocols rely on AHE and costly garbled circuits. Uzun et al. [23] construct LFPSI for Hamming distance based on fully homomorphic encryption (FHE), another costly technique. Using vector oblivious linear evaluation, Chakraborti et al. [4] propose an efficient FPSI for Hamming distance of which cost is independent of d, but at the cost of a non-negligible false positive rate. In addition, they propose an efficient FPSI for L_1 distance in one-dimensional space with the concept of prefix matching. These protocols always perform a brute-force search over all $m \cdot n$ pairs of inputs from both parties, which results in an $m \cdot n$ explosion in communication and computation complexities. In 2024, Chongchitmate et al. [8] propose the most efficient FPSI for Hamming distance to date. Their protocol reduces the $m \cdot n$ explosion through approximating FPSI result via multiple rounds of PSI on sampled components of points. However, its complexities still fail to achieve strict linearity with m and n. Moreover, same with previous protocols in [4,23], they only consider Hamming distance over $\mathbb{F}_2$.

FPSI for $L_{\mathsf{p}\in[1,\infty]}$ Distance. In 2022, Garimella et al. [13] construct the first FPSI protocols for L_1 and L_∞ distance, which are considered as instances of structure-aware PSI in their opinion. For FPSI, their key innovation lies in the use of spatial hashing technique to decrease communication complexity. However, they do not discuss FPSI for general L_p distance and lack the improvement in computation cost. In 2024, Baarsen and Pu [1] propose the first FPSI protocol supporting general $L_{\mathsf{p}\in[1,\infty]}$ distance. They use spatial hashing or similar techniques for coarse filtration on all pairs of both inputs, and propose a novel fuzzy matching protocol based on OKVS and DDH tuple for refined filtering to complete FPSI. Additionally, they go further in protecting Sender privacy by proposing and constructing FPSI-SP. Although many techniques are employed to optimize complexity, complexities of their protocols still remain super-linear factors in n or d, which make their efficiency suffer greatly.

Remark 1. Note that recent protocols are always based on assumptions. It is necessary to introduce assumptions for making costs strictly linear with m and n. The motivation is to limit the number of point pairs that might successfully match in FPSI. If no restrictions are imposed, the number of point pairs that need to be checked is $m \cdot n$, which inevitably leads to an $m \cdot n$ factor in complexities [8].

Table 1. Asymptotic complexities of semi-honest secure FPSI protocols, where Sender holds m points and Receiver holds n points in a d-dimensional space. M is the larger one of m and n. δ is the threshold of FPSI. B_1 and B_2 are parameters in FHE scheme. $\rho \in (0,1)$ is a parameter in LSH scheme. We ignore multiplicative factors of the computational security parameter κ and statistical security parameter λ.

Distance	Protocol	Assumption	Communication	Computation	
				Sender	**Receiver**
Hamming	**[26]**	–	$\mathcal{O}\left(d^2mn\right)$	$\mathcal{O}\left(\mathsf{poly}(d)mn\right)$	$\mathcal{O}\left(d^2mn\right)$
	[23]$^{\ominus}$	FPR&FNR	$\mathcal{O}\left(B_1dmn\right)$	$\mathcal{O}\left(B_2dmn\right)$	$\mathcal{O}\left(\binom{d}{\delta}n\right)$
	[4]$^{\ominus}$	FNR	$\mathcal{O}\left(\delta^2mn\right)$	$\mathcal{O}\left((d+\delta^2)mn\right)$	$\mathcal{O}\left((d+\delta)mn\right)$
	[8]$^{\ominus}$	$\mathsf{R} \wedge \mathsf{S}$. cluster.	$\mathcal{O}\left(dM\log M\right)$	$\mathcal{O}\left(dM\log M\right)$	$\mathcal{O}\left(dM\log M\right)$
	Ours	R. UniqC	$\mathcal{O}\left(d^2m+\delta dn\right)$	$\mathcal{O}\left(d^2m\right)$	$\mathcal{O}\left(d^2m+\delta dn\right)$
$\boldsymbol{L_\infty}$	**[13]**	R. $l_{min} > 2\delta$	$\mathcal{O}\left(m+(4\log\delta)^dn\right)$	$\mathcal{O}\left((2\log\delta)^dm\right)$	$\mathcal{O}\left((2\delta)^dn\right)$
	[1]	R. $l_{min} > 2\delta$	$\mathcal{O}\left(2^dm+\delta dn\right)$	$\mathcal{O}\left(2^ddm\right)$	$\mathcal{O}\left(2^dm+\delta dn\right)$
		R. disj. proj.	$\mathcal{O}\left(m+(\delta d)^2n\right)$	$\mathcal{O}\left(d^2m\right)$	$\mathcal{O}\left(m+(\delta d)^2n\right)$
	Ours	$\mathsf{R} \wedge \mathsf{S}$. disj. proj.	$\mathcal{O}\left(\delta dm+\delta dn\right)$	$\mathcal{O}\left(\delta dm+n\right)$	$\mathcal{O}\left(m+\delta dn\right)$
$\boldsymbol{L_\mathsf{p}}$	**[1]**	R. $l_{min} > 2\delta\left(d^{\frac{1}{\mathsf{p}}}+1\right)$	$\mathcal{O}\left(\delta^{\mathsf{p}}m+\delta 2^ddn\right)$	$\mathcal{O}\left((d+\delta^{\mathsf{p}})m\right)$	$\mathcal{O}\left(m+\delta 2^ddn\right)$
		R. $l_{min} > \frac{1}{\rho}\delta$	$\mathcal{O}\left((\delta^{\mathsf{p}}n^\rho\log n)m+\delta dn^{\rho+1}\right)$	$\mathcal{O}\left(((d+\delta^{\mathsf{p}})n^\rho\log n)m\right)$	$\mathcal{O}\left((n^\rho\log n)m+\delta dn^{\rho+1}\right)$
	Ours	$\mathsf{R} \wedge \mathsf{S}$. disj. proj.	$\mathcal{O}\left((\delta d+\mathsf{p}\log\delta)m+\delta dn\right)$	$\mathcal{O}\left((\delta d+\mathsf{p}\log\delta)m+n\right)$	$\mathcal{O}\left(\mathsf{p}\log\delta m+\delta dn\right)$

- $\ominus$ means that this protocol only handles with Hamming distance on bit vectors.

- FPR (FNR) means that Receiver can tolerate a non-negligible false positive rate (false negative rate).

- $\mathsf{R} \wedge \mathsf{S}$. cluster. means that for both Sender's set and Receiver's set, the Hamming distance between any two points in the same set should be less than δ or greater than $\delta\log n$.

- R. UniqC means that for each Receiver's point, there exists at least $\delta+1$ dimensions such that on each of them this point's component is different from others.

- R. $l_{\min} > l_*$ means that the minimum distance between points of Receiver is greater than l_*.

- R. disj. proj. means that for each Receiver's point, there exists at least one dimension on which its component keeps a distance greater than 2δ from other Receiver's points.

- $\mathsf{R} \wedge \mathsf{S}$. disj. proj. means that the disj. proj. assumption should hold for both Sender's set and Receiver's set.

2 Overview of Our Techniques

In this section, we present a high-level technical overview of our work. And the ideal functionalities for FPSI and its several variants considered in our work are given in Fig. 1.

PARAMETERS: Sender $\mathcal{S}$, Receiver $\mathcal{R}$; Set size m, n; Dimension d; Distance function $\mathsf{dist}(\cdot,\cdot)$, Distance threshold δ; Leakage function $\mathsf{leakage}(\cdot,\cdot)$; Label length σ.

FUNCTIONALITY:

- Wait an input $\mathbf{Q} \in \mathbb{U}^{d\times m}$ from $\mathcal{S}$.
  For LFPSI, wait another input $\mathsf{Label}_{\mathbf{Q}} \in \{0,1\}^{\sigma\times m}$ from $\mathcal{S}$.
- Wait an input $\mathbf{W} \in \mathbb{U}^{d\times n}$ from $\mathcal{R}$.
- Return $\mathsf{leakage}(\mathbf{Q},\mathbf{W})$ to $\mathcal{R}$.

LEAKAGE FUNCTIONS: $\mathsf{leakage}(\mathbf{Q},\mathbf{W})$ is defined as:

- **FPSI:** $\mathsf{leakage}(\mathbf{Q},\mathbf{W}) = \{\mathbf{q}_j \mid \exists\, i \in [n], \mathsf{dist}(\mathbf{q}_j,\mathbf{w}_i) \le \delta\}$.
- **LFPSI:** $\mathsf{leakage}(\mathbf{Q},\mathbf{W}) = \{\mathsf{label}_j \mid \exists\, i \in [n], \mathsf{dist}(\mathbf{q}_j,\mathbf{w}_i) \le \delta\}$, where label_j is the label associated with $\mathbf{q}_j$.
- **FPSI-card:** $\mathsf{leakage}(\mathbf{Q},\mathbf{W}) = \sum_{j\in[m],i\in[n]}(\mathsf{dist}(\mathbf{q}_j,\mathbf{w}_i) \le \delta)$.
- **FPSI-SP:** $\mathsf{leakage}(\mathbf{Q},\mathbf{W}) = \{\mathbf{w}_i \mid \exists\, j \in [m], \mathsf{dist}(\mathbf{q}_j,\mathbf{w}_i) \le \delta\}$.

Fig. 1. Ideal Functionalities for FPSI and Its Variants: $\mathcal{F}_{\mathsf{FPSI}}$, $\mathcal{F}_{\mathsf{LFPSI}}$, $\mathcal{F}_{\mathsf{FPSI-card}}$, and $\mathcal{F}_{\mathsf{FPSI-SP}}$

2.1 Challenge in Efficient FPSI

Most FPSI protocols [1,4,7,11,13,15,23,26], including ours, are based on the same idea: perform FPSI using a batch of fuzzy matching, which can determine whether a Sender's point $\mathbf{q}_j$ and a Receiver's point $\mathbf{w}_i$ satisfy $\mathsf{dist}(\mathbf{q}_j,\mathbf{w}_i) \le \delta$, and return the result to Receiver. Therefore, there are two directions for improving FPSI protocols: one is the optimization of fuzzy matching protocol, which is the focus of most existing works [4,7,11,15,23,26], and the other is the optimization of the process of reducing FPSI to fuzzy matching, which is the focus of this work.

The above FPSI paradigm can be decomposed into two phases: "coarse mapping" and "refined filtering" [1]. Coarse mapping is used to assign several identifiers to points of Sender and Receiver, and two points from Sender and Receiver respectively with a same identifier will form a pair[2]. Refined filtering is used to perform fuzzy matching on each pair obtained from coarse mapping to get the final result.

The main complexity bottlenecks of existing works are derived from their coarse mapping methods. For example, the naive coarse mapping which brutally traverses all pairs of inputs from parties results in an unacceptable $m \cdot n$

[2] A point can appear in multiple pairs.

blowup in complexities. Besides, [13] uses spatial hashing technique to perform coarse mapping. In this coarse mapping, a Receiver's point is mapped to $\mathcal{O}(2^d)$ identifiers. This expansion is the source of the factor 2^d in complexities.

The challenge in efficient FPSI protocols is to construct coarse mapping methods with minor expansion on input sizes to break bottlenecks.

2.2 Fuzzy Mapping

We abstract the coarse mapping into a new cryptographic primitive named fuzzy mapping (Fmap), with the complexity bottleneck being formalized as the expansion rate of Fmap. As Sect. 4.2 will demonstrate, almost all known FPSI protocols are constructed based on instances of Fmap. Thus, proposing non-trivial Fmap instances is the core task in this work.

The input of Fmap consists of m points $(\mathbf{q}_j)_{j\in[m]} \in \mathbb{U}^{d\times m}$ from Sender and n points $(\mathbf{w}_i)_{i\in[n]} \in \mathbb{U}^{d\times n}$ from Receiver. The output of Fmap consists of $\left(\mathsf{ID}\left(\mathbf{q}_j\right)\right)_{j\in[m]}$ for Sender and $\left(\mathsf{ID}\left(\mathbf{w}_i\right)\right)_{i\in[n]}$ for Receiver, where $\mathsf{ID}\left(\mathbf{q}\right)$ and $\mathsf{ID}\left(\mathbf{w}\right)$ are subsets of an identifier universe $\mathscr{I}$.

Three Requirements. For realizing the functionality of coarse mapping securely, Fmap for $\mathsf{dist}\left(\cdot,\cdot\right)$ of threshold δ should satisfy the following requirements:

- $\mathsf{ID}\left(\mathbf{q}_j\right)$ should intersect with $\mathsf{ID}\left(\mathbf{w}_i\right)$ when $\mathsf{dist}\left(\mathbf{q}_j, \mathbf{w}_i\right)$ is not greater than δ. Otherwise, coarse mapping would lose the pair $(\mathbf{q}_j, \mathbf{w}_i)$, leading to an incorrect FPSI result. Note that the existence of refined filtering allows Fmap to tolerate false positives[3].
- The probability that there exist two distinct points $\mathbf{w}_i$ and $\mathbf{w}_{i'}$ in $\mathbf{W}$ such that $\mathsf{ID}\left(\mathbf{w}_i\right)$ intersects with $\mathsf{ID}\left(\mathbf{w}_{i'}\right)$ is negligible. Otherwise, an identifier might lead to point pairs involving multiple Receiver's points, which could lead to incomplete executions of fuzzy matching in refined filtering[4].
- For security, Fmap should not reveal any information about one party's input to the other party. In other words, the view of Receiver invoking Fmap with Sender should be computationally indistinguishable from that with another Sender, and the same applies to Sender's perspective.

Expansion Rate. We define the Sender's expansion rate and Receiver's expansion rate of Fmap as the ratio of the output size to the input size for Sender and Receiver respectively.

It is clear that the optimal expansion rate of Fmap is 1. We use unit Fmap (UFmap) to denote the Fmap with both expansion rates of 1, and unit Fmap for Sender (sUFmap) to denote the Fmap with Sender's expansion rate of 1.

[3] That is to say, cases that $\mathsf{ID}\left(\mathbf{q}_j\right)$ intersects with $\mathsf{ID}\left(\mathbf{w}_i\right)$ and $\mathsf{dist}\left(\mathbf{q}_j, \mathbf{w}_i\right)$ is greater than δ are allowed.

[4] In order to hide the distribution of points, Receiver can only initiate fuzzy matching once for each identifier. If multiple fuzzy matchings are performed on an identifier, Sender can infer that there are multiple Receiver's points nearby.

2.3 Non-trivial Fmap for Hamming Distance

There is no Fmap instance for Hamming distance except the naive one (i.e. brutally traversing all $m \cdot n$ pairs of input points), which is the primary culprit for $m \cdot n$ blowup in complexity. Therefore, we hope to find a non-trivial Fmap to improve FPSI for Hamming distance.

We assume that each Receiver's point has at least $\delta + 1$ unique components, and denote this assumption as Receiver's unique components (R. UniqC) assumption. Typically, $\delta \ll d$ holds for applications of FPSI [18,19]. Thus, R. UniqC assumption is intuitively reasonable, and furthermore, we formally prove that it holds with overwhelming probability for uniformly random Receiver's input in Sect. 5.1. R. UniqC assumption reflects real-world scenarios where legitimate texts or numbers vary significantly, and their errors are merely deviations of a few characters, while Receiver hopes to query several entries under such circumstances [8].

Under R. UniqC assumption, a non-trivial Fmap for Hamming distance, which we refer to as UniqC Fmap, can be constructed, where Receiver's points are mapped to their unique components, while each Sender's point is mapped to all of its d components. More details about UniqC Fmap are shown in Sect. 5.1.

2.4 UFmap for L_∞ Distance

UFmap for L_∞ Distance is Enough. To overcome complexity bottlenecks in FPSI protocols for $L_{\mathsf{p}\in[1,\infty]}$ distance, we hanker for a UFmap for $L_{\mathsf{p}\in[1,\infty]}$ distance. Fortunately, benefiting from the facts that Fmap can tolerate false positives and that the L_∞ distance between any two points is always no greater than the $L_{\mathsf{p}\in[1,\infty]}$ distance, we can use Fmap for L_∞ distance in FPSI for general $L_{\mathsf{p}\in[1,\infty]}$ distance. Therefore, we will only discuss the construction of UFmap for L_∞ distance in the following paragraphs.

A Toy Protocol from Spatial Additive Sharing. Let us first consider a toy protocol in a simplified setting where Receiver chooses $\mathsf{seed}_{\mathbf{w},\mathcal{R}}$ for $\mathbf{w} \in \mathbf{W}$ as an assignment and Sender wants to choose $\mathsf{seed}_{\mathbf{q},\mathcal{R}}$ as the assignment of $\mathbf{q} \in \mathbf{Q}$ meeting $\mathsf{seed}_{\mathbf{q},\mathcal{R}}$ equals to $\mathsf{seed}_{\mathbf{w},\mathcal{R}}$ when $L_\infty(\mathbf{q}, \mathbf{w})$ is not greater than δ.

The rough idea is to share assignment $\mathsf{seed}_{\mathbf{w},\mathcal{R}}$ of Receiver's point $\mathbf{w}$ via additive secret sharing across those positions close to its components on d dimensions as their assignments[5], and then have Sender reconstruct the point's assignment using shares from each dimension. This idea is termed as *spatial additive sharing* (SAS).

Certainly, Receiver's assignments at these positions (Receiver's assigned coordinate system), should not be obtained in plaintext by Sender, or Sender will know which is the component of $\mathbf{w}$ by comparing whether two adjacent positions were assigned the same shares, which violates security. Therefore, Receiver should use AHE to hide the assigned coordinate system.

[5] There are $2\delta + 1$ positions centered around each component and their $2\delta + 1$ assignments are the same secret share.

Conversion from Toy Protocol to UFmap. The toy protocol satisfies the first requirement of Fmap in simplified setting. For crossing the gap between it and UFmap, we should enhance the design to fully meet all three requirements.

By introducing the assumption from [1] that each Receiver's point maintains a distance of more than 2δ on at least one dimension from the others, we can ensure that each Receiver's point has at least a share that is independent of the others. Consequently, the second requirement is satisfied.

Moreover, Sender can perform exactly the same as Receiver, including assigning values to coordinate system and points. Thus, if the same assumption also holds for Sender's input, each assignment of Sender's point in own assigned coordinate system is also imported with at least one independently uniform random share. In order to prevent the final result from being used to deduce the assignment of one's own point in the opponent's assigned coordinate system, we additionally embed a Diffie-Hellman (DH) subprotocol.

In summary, a point's ID from this Fmap contains only one element called id, which is the sum, protected by the DH keys of both parties, of assignments of the point in assigned coordinate systems of both parties. Building on the above idea, we construct a UFmap for L_∞ distance, which we call SAS Fmap. Since its expansion rate is 1, SAS Fmap is capable of circumventing complexity bottlenecks in FPSI protocols for $L_{\mathsf{p}\in[1,\infty]}$ distance.

2.5 Applications of Fmap

mqFRPMT from sUFmap. Chen et al. [6] demonstrate the powerful capabilities of mqRPMT as a central block in their private set operation (PSO) framework. An attractive idea is to use the fuzzy version of mqRPMT to provide a unified framework for FPSI and its variants. Thus, we propose multi-query fuzzy RPMT (mqFRPMT).

Roughly speaking, mqFRPMT is a two-party protocol between Sender holding $\mathbf{Q} = (\mathbf{q}_1, \cdots, \mathbf{q}_m)$ and Receiver holding $\mathbf{W} = (\mathbf{w}_1, \cdots, \mathbf{w}_n)$. After invoking of mqFRPMT for distance $\mathsf{dist}(\cdot, \cdot)$ and threshold δ, Receiver learns an indication bit vector $\mathbf{e} = (e_1, \cdots, e_m) \in \{0, 1\}^m$ such that e_i equals to 1 if and only if there exists a point $\mathbf{w}_j \in \mathbf{W}$ meeting $\mathsf{dist}(\mathbf{q}_i, \mathbf{w}_j)$ is not greater than δ, while Sender learns nothing.

We present a generic construction of mqFRPMT from sUFmap, OKVS, and fuzzy matching. Firstly, Receiver and Sender invoke sUFmap to get identifiers for their points. The first requirement of sUFmap guarantees that a Sender's point and a Receiver's point have a same identifier when they are close enough. For each Receiver's point, Receiver generates keys with the point's identifiers, and uses the message required to execute fuzzy matching with this point as value. Using these key-value pairs, Receiver encodes an OKVS and sends it to Sender. Sender decodes the OKVS using keys from identifiers of Sender's points and continues to execute fuzzy matching, which will eliminate false positives in sUFmap result, ultimately allowing Receiver to obtain the result of mqFRPMT.

FPSI from Fmap. Consider a general Fmap that might not be an sUFmap. For each point of Sender, Receiver obtains multiple fuzzy matching results instead of one, and the number of 1 in these results may reveal additional information about this Sender's point to Receiver, violating the security of mqFRPMT.

However, if the ultimate goal is to construct FPSI, this leakage will not affect the security[6]. Thus, any Fmap can be utilized to construct the corresponding FPSI using a generic method, while only sUFmap can directly yield mqFRPMT[7].

2.6 Applications of mqFRPMT

With oblivious transfer (OT), we can derive FPSI, LFPSI, and FPSI-card from mqFRPMT by adopting the exact same approaches as that from mqRPMT to obtain PSI, LPSI, and PSI-card.

Special Variant FPSI-SP from mqFRPMT and UFmap. As an exception, FPSI-SP cannot be simply realized by replicating the framework of PSI because of its asymmetry. We observe that, Sender can obtain unique identifiers of Receiver's points in FPSI-SP result from UFmap. Therefore, if Sender uses the result of UFmap as points' labels, Receiver can learn the corresponding identifiers of points in FPSI-SP result by invoking LFPSI with Sender. At last, Receiver can trace back to get the result of FPSI-SP with these identifiers.

Figure 2 gives a pictorial overview of our work.

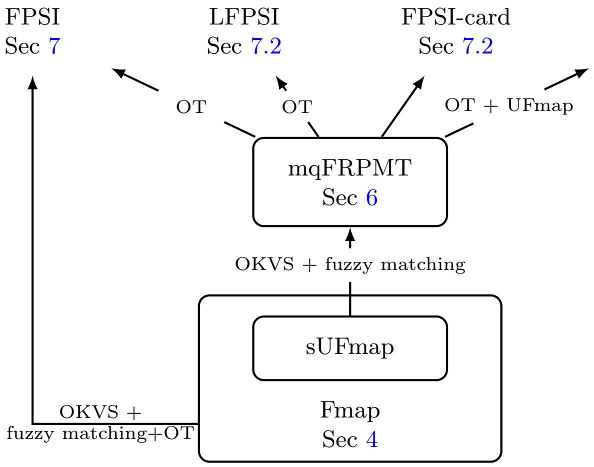


Fig. 2. Summary of our work. The rectangles denote notions newly in this work.

[6] Because FPSI allows Receiver to obtain information about these points.

[7] As will be shown later, using fuzzy matching that outputs secret shares can solve this problem.

3 Preliminaries

For lack of space, we put Additively Homomorphic Encryption, Oblivious Transfer, and Semi-Honest Security Model in the full version.

3.1 Notation

We use κ, λ to denote the computational and statistical security parameters respectively. We use $[n]$ to denote the set $\{1, 2, \cdots, n\}$ and $[n, m]$ to denote the set $\{n, n+1, \cdots, m\}$. We assume that every set X of size $|X|$ has a default order (e.g. lexicographical order), and represent it as $X = \left(x_1, \cdots, x_{|X|}\right) = (x_i)_{i \in [|X|]}$. We use $\leftarrow$ to denote assignment and $x \overset{\mathsf{R}}{\leftarrow} X$ to denote sampling x uniformly at random from X. A function is negligible in ℓ, written $\mathsf{negl}(\ell)$, if it vanishes faster than the inverse of any polynomial in ℓ. $\boldsymbol{x} \| \boldsymbol{y}$ is the concatenation of two strings $\boldsymbol{x}$ and $\boldsymbol{y}$. For a key-value pairs multiset List, we use $\mathsf{List}[k]$ to denote the value for key k.

For parameters in FPSI, we use d to denote the dimension of space, δ to denote the threshold, and $\mathsf{dist}(\cdot, \cdot)$ to denote the distance function. We use $\mathcal{H}$, L_∞, and L_p distance as Hamming, infinite norm, and Minkowski distance, respectively. To simplify the statement, we use $L_{\mathsf{p} \in [1,\infty]}$ to represent the union of $L_{\mathsf{p} \in [1,\infty)}$ and L_∞. We use $\mathcal{S}$ to denote Sender, who holds set $\mathbf{Q} \in \mathbb{U}^{d \times m}$ of size m, and $\mathcal{R}$ to denote Receiver, who holds set $\mathbf{W} \in \mathbb{U}^{d \times n}$ of size n, where d is the dimension. Here we use 2^u to denote the size of alphabet $\mathbb{U}$. We use $\mathbf{q}_j$ and $\mathbf{w}_i$ as points in $\mathbf{Q}$ and $\mathbf{W}$ respectively. $q_{j,k}$ represents the component of point $\mathbf{q}_j$ on dimension k, and $w_{i,k}$ is analogous. $\mathsf{ball}_{\mathbf{w}_i}^{\mathsf{dist}(\cdot,\cdot)}$ represents a d-dimensional ball with $\mathbf{w}_i$ as center and δ as radius. We use $\mathscr{I}$ to denote the identifier universe. $\mathsf{ID}(\mathbf{q}_j), \mathsf{ID}(\mathbf{w}_i) \subset \mathscr{I}$ are sets of $\mathbf{q}_j$'s identifiers and $\mathbf{w}_i$'s identifiers respectively.

Specifically, for any two points $\mathbf{q}, \mathbf{w} \in \mathbb{U}^d$, Hamming distance over $\mathbb{U}$ is $\mathcal{H}_\mathbb{U}(\mathbf{q}, \mathbf{w}) = \mathcal{H}(\mathbf{q}, \mathbf{w}) = \sum_{k=1}^{d} (q_k \neq w_k)$, and Hamming distance over $\mathbb{U}^P$ is $\mathcal{H}_{\mathbb{U}^P}(\mathbf{q}, \mathbf{w}) = \sum_{k'=0}^{\frac{d}{P}-1} (\widetilde{q_{k'}} \neq \widetilde{w_{k'}})$, where $\widetilde{q_{k'}} = q_{k' \cdot P+1} \| q_{k' \cdot P+2} \| \cdots \| q_{k' \cdot P+P}$ and $\widetilde{w_{k'}} = w_{k' \cdot P+1} \| w_{k' \cdot P+2} \| \cdots \| w_{k' \cdot P+P}$.

3.2 Oblivious Key-Value Store

The oblivious key-value store (OKVS) is a data structure consisting of Encode and Decode algorithms that enables encoding n key-value pairs such that an adversary can not infer the original input keys with the encoding result, when the input values are random [12].

In addition, our Fmap and mqFRPMT protocols require independence property for OKVS, which means decoding a non-encoded key will yield a uniformly random result. Bienstoc et al. [2] prove that their RB-OKVS satisfies independence property[8].

[8] They call this property "random decoding".

The formal definitions of OKVS and its independence property are given in the full version.

For evaluating the efficiency of OKVS, there are typically three measures: rate, encoding cost, and decoding cost. The rate is the ratio between number n of input pairs and output size m. Recent OKVS constructions [2,12,21] achieve constant rate, $\mathcal{O}(n\lambda)$ encoding cost, and $\mathcal{O}(\lambda)$ decoding cost.

3.3 Fuzzy Matching

Fuzzy matching enables Sender and Receiver determine whether the Sender's point $\mathbf{q}$ and the Receiver's point $\mathbf{w}$ satisfy $\mathsf{dist}(\mathbf{q}, \mathbf{w}) \leq \delta$ [1]. Its functionality is given in Fig. 3. Obviously, let the protocol return the result by secret shares, and we get secret-shared fuzzy matching.

Our work is concerned with Hamming and $L_{\mathsf{p}\in[1,\infty]}$ distances. For Hamming distance, considering points of two parties as their Boolean shares in the case $\mathbb{U} = \{0, 1\}$, there is a trivial approach of (secret-shared) fuzzy matching that consists of OT-based conversion of Boolean sharing to Arithmetic sharing and (secret-shared) secure comparing [20]. This approach has $\mathcal{O}(d)$ communication and computation costs. For $L_{\mathsf{p}\in[1,\infty]}$ distance, Baarsen and Pu provide constructions of fuzzy matching in [1].

PARAMETERS: Sender $\mathcal{S}$, Receiver $\mathcal{R}$; Dimension d; Distance function $\mathsf{dist}(\cdot, \cdot)$; Distance threshold δ.

FUNCTIONALITY:

- Wait an input $\mathbf{q} \in \mathbb{U}^d$ from $\mathcal{S}$.
- Wait an input $\mathbf{w} \in \mathbb{U}^d$ from $\mathcal{R}$.
- Return $e \in \{0, 1\}$ to $\mathcal{R}$, where $e = 1$ if and only if $\mathsf{dist}(\mathbf{q}, \mathbf{w}) \leq \delta$.

Fig. 3. Ideal Functionality for Fuzzy Matching $\mathcal{F}_{\mathsf{FMatch}}$

One of building blocks we use is a special case of fuzzy matching, fuzzy matching for interval (IFmat), by which Sender with an interval and Receiver with a number can check whether this number belongs to the interval. Moreover, if δ is set to 0, this special case of IFmat is private equality test (PEqT). Their functionalities are given in Fig. 4.

Using the idea of prefix matching, Chakraborti et al. [4] propose a semi-honest secure IFmat protocol achieving communication and computation complexities scaling logarithmically in the threshold. In all our constructions, we will use their protocol to instantiate IFmat and PEqT.

4 Fuzzy Mapping

In this section, we provide the formal definitions for fuzzy mapping (Fmap) and its expansion rate, and list existing instances of Fmap.

PARAMETERS: Sender $\mathcal{S}$, Receiver $\mathcal{R}$; Threshold δ.

FUNCTIONALITY:

- Wait an input $a \in \mathbb{Z}$ from $\mathcal{S}$.
- Wait an input $x \in \mathbb{Z}$ from $\mathcal{R}$.
- Return e to $\mathcal{R}$, where $e = 1$ if and only if:
  IFmat: $x \in [a-\delta, a+\delta]$
  PEqT: $x = a$

Fig. 4. Ideal Functionalities for IFmat $\mathcal{F}_{\mathsf{IFmat}}$ and PEqT $\mathcal{F}_{\mathsf{PEqT}}$

4.1 Definition of Fmap

As mentioned in Sect. 2.2, with Fmap, both parties can map each of their points to a set of identifiers. If a Sender's point and a Receiver's point are close enough, they will have a same identifier, and point pairs formed in this way will be further filtered by fuzzy matching to obtain FPSI result.

The formal definition of Fmap is as follows.

Definition 1 (Fuzzy Mapping). *A two-party protocol Π, where Sender's input $\mathbf{Q} = (\mathbf{q}_j)_{j\in[m]} \in \mathbb{U}^{d\times m}$ results in $\mathsf{ID}(\mathbf{Q}) = \big(\mathsf{ID}(\mathbf{q}_j)\big)_{j\in[m]}$ and Receiver's input $\mathbf{W} = (\mathbf{w}_i)_{i\in[n]} \in \mathbb{U}^{d\times n}$ results in $\mathsf{ID}(\mathbf{W}) = \big(\mathsf{ID}(\mathbf{w}_i)\big)_{i\in[n]}$[9], is a semi-honest secure* fuzzy mapping (Fmap) *protocol $\Pi^{\mathsf{dist}(\cdot,\cdot)}_{\mathsf{Fmap}}$ of threshold δ for* $\mathsf{dist}(\cdot,\cdot)$, *if and only if Π satisfies:*

- **Correctness.** *For any two points $\mathbf{q}_j \in \mathbf{Q}$ and $\mathbf{w}_i \in \mathbf{W}$:*

$$\mathsf{dist}(\mathbf{q}_j, \mathbf{w}_i) \leq \delta \Longrightarrow \mathsf{ID}(\mathbf{q}_j) \cap \mathsf{ID}(\mathbf{w}_i) \neq \emptyset$$

- **Distinctiveness.** *For the output $\mathsf{ID}(\mathbf{W})$ of Receiver, the following equation holds:*

$$\Pr[\exists\ i, i' \in [n], s.t. (i \neq i') \wedge (\mathsf{ID}(\mathbf{w}_i) \cap \mathsf{ID}(\mathbf{w}_{i'}) \neq \emptyset)] = \mathsf{negl}(\kappa)$$

- **Security.** *Considering corrupt semi-honest Sender, for any $\mathbf{Q} \in \mathbb{U}^{d\times m}$ and any $\mathbf{W}, \mathbf{W}' \in \mathbb{U}^{d\times n}$, it holds that*

$$\mathsf{view}^{\Pi}_{\mathcal{S}}(\kappa, \lambda; \mathbf{Q}, \mathbf{W}) \overset{c}{\approx} \mathsf{view}^{\Pi}_{\mathcal{S}}(\kappa, \lambda; \mathbf{Q}, \mathbf{W}')$$

  Considering corrupt semi-honest Receiver, for any $\mathbf{W} \in \mathbb{U}^{d\times n}$ and any $\mathbf{Q}, \mathbf{Q}' \in \mathbb{U}^{d\times m}$, it holds that

$$\mathsf{view}^{\Pi}_{\mathcal{R}}(\kappa, \lambda; \mathbf{Q}, \mathbf{W}) \overset{c}{\approx} \mathsf{view}^{\Pi}_{\mathcal{R}}(\kappa, \lambda; \mathbf{Q}', \mathbf{W})$$

To quantify the expansion of inputs, we define the expansion rate of Fmap.

[9] $\mathsf{ID}(\mathbf{q}_j), \mathsf{ID}(\mathbf{w}_i) \subset \mathscr{I}$; for security reason, we default to $|\mathsf{ID}(\mathbf{q}_j)| = |\mathsf{ID}(\mathbf{q}_{j'})|$ for different $j, j' \in [m]$ and $|\mathsf{ID}(\mathbf{w}_i)| = |\mathsf{ID}(\mathbf{w}_{i'})|$ for different $i, i' \in [n]$.

Definition 2 (Expansion Rate). *The expansion rate of Fmap for Sender's input is*

$$\mathsf{rate}_{\mathcal{S}} = \frac{1}{m} \sum_{j \in [m]} \left| \mathsf{ID}\left(\mathbf{q}_j\right) \right|$$

The expansion rate of Fmap for Receiver's input is

$$\mathsf{rate}_{\mathcal{R}} = \frac{1}{n} \sum_{i \in [n]} \left| \mathsf{ID}\left(\mathbf{w}_i\right) \right|$$

The expansion rate of Fmap is

$$\mathsf{rate} = \max\left\{\mathsf{rate}_{\mathcal{S}}, \mathsf{rate}_{\mathcal{R}}\right\}$$

Definition 3 (Sender's Unit Fmap). *An Fmap is a* Sender's unit Fmap (sUFmap) *if and only if its expansion rate for Sender's input is 1.*

Definition 4 (Unit Fmap). *An Fmap is a* unit Fmap (UFmap) *if and only if its expansion rate is 1.*

The efficiency of an Fmap instance is measured by:

- **Expansion rate:** Expansion rate of Fmap is positively related to complexities of FPSI based on it, thus we hope it to be as small as possible. Note that the optimal expansion rate is 1.
- **Communication complexity:** As a two-party protocol, Fmap's own efficiency is influenced by its communication complexity. Since many Fmap instances degenerate into two algorithms executed by Sender and Receiver respectively, they have no communication.
- **Computation complexity:** The computation complexity of Fmap is also a factor to consider, and it is clear that the lower bound of computation complexity is the size of output.

A crucial observation is that as long as the complexity of Fmap does not exceed that of the subsequent part, the asymptotic complexity of the entire FPSI will not be affected.

Therefore, a high-level intuition is that we can improve the overall efficiency of FPSI by reducing expansion rate of Fmap at the cost of a tolerable increase in complexity of Fmap.

Lemma 1 (Reduction of Fmap). *If there are two distance functions* $\mathsf{dist}(\cdot,\cdot)$ *and* $\mathsf{dist}'(\cdot,\cdot)$ *such that* $\mathsf{dist}(\mathbf{q},\mathbf{w}) \le \mathsf{dist}'(\mathbf{q},\mathbf{w})$ *holds for any two points* $\mathbf{q}$ *and* $\mathbf{w}$*, then Fmap protocol* $\Pi_{\mathsf{Fmap}}^{\mathsf{dist}(\cdot,\cdot)}$ *realizes Fmap for* $\mathsf{dist}'(\cdot,\cdot)$*.*

Proof. As Fmap for $\mathsf{dist}'(\cdot,\cdot)$, the distinctiveness and security of $\Pi_{\mathsf{Fmap}}^{\mathsf{dist}(\cdot,\cdot)}$ are guaranteed by Definition 1 of Fmap for $\mathsf{dist}(\cdot,\cdot)$.

Now consider the correctness of $\Pi_{\mathsf{Fmap}}^{\mathsf{dist}(\cdot,\cdot)}$ for $\mathsf{dist}'(\cdot,\cdot)$. For any $j \in [m]$ and $i \in [n]$, when $\mathsf{dist}'(\mathbf{q}_j,\mathbf{w}_i) \le \delta$. We have $\mathsf{dist}(\mathbf{q}_j,\mathbf{w}_i) \le \mathsf{dist}'(\mathbf{q}_j,\mathbf{w}_i) \le \delta$. Thus the correctness for $\mathsf{dist}'(\cdot,\cdot)$ is guaranteed by the correctness for $\mathsf{dist}(\cdot,\cdot)$.

Therefore, $\Pi_{\mathsf{Fmap}}^{\mathsf{dist}(\cdot,\cdot)}$ realizes Fmap for $\mathsf{dist}'(\cdot,\cdot)$.

Corollary 1. *For any $P \in \mathbb{N}^+$ and any two points $\mathbf{q}, \mathbf{w} \in \mathbb{U}^d$, we have $\mathcal{H}_{\mathbb{U}^P}(\mathbf{q}, \mathbf{w}) \leq \mathcal{H}_{\mathbb{U}}(\mathbf{q}, \mathbf{w})$. According to Lemma 1, $\Pi_{\mathsf{Fmap}}^{\mathcal{H}_{\mathbb{U}^P}}$ can be seen as $\Pi_{\mathsf{Fmap}}^{\mathcal{H}_{\mathbb{U}}}$.*

Corollary 2. *For any two points $\mathbf{q}, \mathbf{w} \in \mathbb{U}^d$, we have $L_\infty(\mathbf{q}, \mathbf{w}) \leq L_{\mathsf{p}\in[1,\infty]}(\mathbf{q}, \mathbf{w})$. According to Lemma 1, $\Pi_{\mathsf{Fmap}}^{L_\infty}$ can be seen as $\Pi_{\mathsf{Fmap}}^{L_{\mathsf{p}\in[1,\infty]}}$.*

4.2 Existing Fmap Constructions

Table 2 lists existing constructions that fit to Definition 1 of Fmap. Many of existing FPSI protocols are constructed using Fmap instances listed in this table.

As can be seen in Table 2, all of previous Fmap instances have communication cost of zero and computation cost of theoretical lower bound but expansion rate of pretty big value, while our UniqC Fmap is the only non-trivial Fmap for Hamming distance and our SAS Fmap achieves the optimal expansion rate but has non-optimal complexities. This trade-off works well because complexity bottlenecks in previous FPSI protocols actually come from expansion rates rather than costs of invoking Fmap.

Table 2. Comparison of Fmap instances, where m and n are set sizes of Sender's and Receiver's inputs, respectively. d is the space dimension. $\rho \in (0, 1)$ is a parameter in LSH scheme. We ignore multiplicative factors of the computational security parameter κ and statistical parameter λ.

Fmap	Distance	$\mathsf{rate}_{\mathcal{S}}$	$\mathsf{rate}_{\mathcal{R}}$	Communication	Computation	
					Sender	Receiver
Naive [11]	**Anyone**	n	1	–	$\mathcal{O}(nm)$	$\mathcal{O}(n)$
Spatial Hashing [13]	L_∞	1	$\mathcal{O}(2^d)$	–	$\mathcal{O}(m)$	$\mathcal{O}(2^d n)$
Separated Balls [1]	L_∞	d	$\mathcal{O}(\delta)$	–	$\mathcal{O}(dm)$	$\mathcal{O}(\delta n)$
LSH [1]	$L_{\mathsf{p}\in[1,\infty)}$	$\mathcal{O}(n^\rho \log n)$	$\mathcal{O}(n^\rho)$	–	$\mathcal{O}((n^\rho \log n) m)$	$\mathcal{O}(n^{\rho+1})$
Ours: UniqC	**Hamming**	d	$\delta + 1$	–	$\mathcal{O}(dm)$	$\mathcal{O}(\delta n)$
Ours: SAS	L_∞	1	1	$\mathcal{O}(\delta dm + \delta dn)$	$\mathcal{O}(\delta dm + n)$	$\mathcal{O}(m + \delta dn)$

- **Naive Fmap.** A straightforward approach of FPSI is to perform fuzzy matching on all pairs of these two inputs to obtain results. This idea can be abstracted into a naive Fmap: for each Sender's point $\mathbf{q}_j$, $\mathsf{ID}(\mathbf{q}_j)$ is $\{i\}_{i\in[n]}$, thus $\mathsf{rate}_{\mathcal{S}}$ is $\mathcal{O}(n)$; for each Receiver's point $\mathbf{w}_i$, $\mathsf{ID}(\mathbf{w}_i)$ is $\{i\}$ where i is the index of this point, thus $\mathsf{rate}_{\mathcal{R}}$ is 1. It does not rely on any assumptions and can be used in FPSI for any distance function. Many existing FPSI protocols [4,7,11,15,23,26] for Hamming distance adopt naive Fmap, which leads to the $m \cdot n$ blowup in their complexities.
- **Prior Non-trivial Fmap.** Recently, some works [1,13] try to avoid the $m \cdot n$ blowup. We abstracted three non-trivial Fmap from them, and the detailed analysis can be found in the full version.
- **New Fmap.** Details of our Fmap instance will be given later.

5 New Fmap Constructions

In this section, we present new semi-honest secure Fmap constructions for Hamming and L_∞ distances, which are the infrastructure for subsequent protocols.

5.1 UniqC Fmap for Hamming Distance

We present a construction of semi-honest secure Fmap for Hamming distance, which is denoted by UniqC Fmap. Similar to existing Fmap constructions, UniqC Fmap consists of two algorithms for Sender and Receiver respectively due to the absence of interaction. For each Receiver's point, Receiver chooses its $\delta+1$ unique components as its ID, while Sender selects all d components of a point as its ID. The formal description of UniqC Fmap is shown in Fig. 5.

PARAMETERS:

– Sender $\mathcal{S}$ and Receiver $\mathcal{R}$.

INPUT OF $\mathcal{S}$: $\mathbf{Q} = (\mathbf{q}_1, \cdots, \mathbf{q}_m) \in \mathbb{U}^{d\times m}$.
INPUT OF $\mathcal{R}$: $\mathbf{W} = (\mathbf{w}_1, \cdots, \mathbf{w}_n) \in \mathbb{U}^{d\times n}$.

$\mathcal{S}$'s $\mathsf{UniqC\ Fmap}(\mathbf{Q})$:

1. For each $j \in [m]$, $\mathcal{S}$ computes $\mathsf{ID}(\mathbf{q}_j) \leftarrow \{k \| q_{j,k}\}_{k\in[d]}$.
2. Return $\mathsf{ID}(\mathbf{Q}) = (\mathsf{ID}(\mathbf{q}_1), \cdots, \mathsf{ID}(\mathbf{q}_m))$.

$\mathcal{R}$'s $\mathsf{UniqC\ Fmap}(\mathbf{W})$:

1. For each $i \in [n]$, $\mathcal{R}$ computes $\mathsf{ID}(\mathbf{w}_i) \leftarrow \left\{u_k^i \| w_{i,u_k^i}\right\}_{k\in[\delta+1]}$, where $u_k^i \in [d]$ is a dimension such that $w_{i,u_k^i} \neq w_{i',u_k^i}$ holds for any $i' \in [n]\setminus\{i\}$.
2. Return $\mathsf{ID}(\mathbf{W}) = (\mathsf{ID}(\mathbf{w}_1), \cdots, \mathsf{ID}(\mathbf{w}_n))$.

Fig. 5. Fmap Protocol for Hamming distance: $\Pi^{\mathcal{H}}_{\mathsf{UniqC\ Fmap}}$

Definition 5 (Unique Set). *For a point* $\mathbf{w} \in \mathbf{W}$ *in a d-dimensional space,* $\mathbf{w}$ *has a* unique component w_k*, if and only if its component on dimension k is different from that of any other point in* $\mathbf{W}$. *A point* $\mathbf{w}$ *is* unique*, if and only if* $\mathbf{w}$ *has at least* $\delta + 1$ *unique components. A set* $\mathbf{W}$ *is* unique*, if and only if all points in* $\mathbf{W}$ *are unique.*

Lemma 2 (Uniform Distribution). *In a d-dimensional space, if points in set* $\mathbf{W}$ *are uniformly distributed, then the probability that* $\mathbf{W}$ *is unique is* $1 - \mathsf{negl}(d)$[10].

[10] Here we default that the size of alphabet $\mathbb{U}$ is greater than n. When $|\mathbb{U}| = 2^u \leq n$, we can pack P components as one super-component such that $\left|\mathbb{U}^P\right| = 2^{uP} > n$, thus R. UniqC assumption for $\mathcal{H}_{\mathbb{U}^P}$ holds and $\Pi^{\mathcal{H}_{\mathbb{U}^P}}_{\mathsf{UniqC\ Fmap}}$ works. According to Corollary 1, we can use $\Pi^{\mathcal{H}_{\mathbb{U}^P}}_{\mathsf{UniqC\ Fmap}}$ as $\Pi^{\mathcal{H}_{\mathbb{U}}}_{\mathsf{UniqC\ Fmap}}$.

Proof. For each $\mathbf{w}_i \in \mathbf{W}$ and each dimension $k \in [d]$, we have

$$\Pr[w_{i,k} \text{ is not a unique component}] \leq \frac{n-1}{2^u}$$

Hence, the probability that $\mathbf{w}_i$ has exactly δ unique components is not greater than $\binom{d}{\delta}\left(\frac{n-1}{2^u}\right)^{d-\delta}$. By a union bound, it holds that

$$\Pr[\mathbf{w}_i \text{ is not unique}] \leq \delta\tbinom{d}{\delta}\left(\frac{n-1}{2^u}\right)^{d-\delta} \leq d^{\delta+1}\left(\frac{n-1}{2^u}\right)^{d-\delta} \triangleq f(d)$$

We default that $2^u > n-1$ and thus $f(d)$ is $\mathsf{negl}(d)$.

$$\Pr[\mathbf{W} \text{ is unique}] = (1 - \Pr[\mathbf{w}_i \text{ is not unique}])^n \geq (1 - f(d))^n$$

which is $1 - \mathsf{negl}(d)$.

Remark 2. Considering 400-dimensional bio-bit-vectors and $\delta = 7$ in [24], we pack 16 bits into a super-component (i.e. d and u are updated to 25 and 16 respectively), and we choose statistical security parameter $\lambda = 40$.

Then, when $n < 2^{11}$, we have

$$\Pr[\mathbf{W} \text{ is unique}] \geq \left(1 - d^{\delta+1}\left(\frac{n-1}{2^u}\right)^{d-\delta}\right)^n \geq 1 - 2^{-\lambda}$$

For UniqC Fmap, we introduce the Receiver's unique components (R. UniqC) assumption:

Each Receiver's point has unique components on at least $\delta+1$ dimensions.

If Receiver's points are uniformly distributed, then according to Lemma 2, R. UniqC assumption holds in high-dimensional case with overwhelming probability. Thus, it is acceptable to base our construction on it. Now, we prove the protocol in Fig. 5 is a semi-honest secure Fmap for Hamming distance.

Theorem 1 (Correctness). *The protocol presented in Fig. 5 satisfies the correctness defined in Definition 1 for Hamming distance.*

Proof. For $\mathbf{q}_j \in \mathbf{Q}$ and $\mathbf{w}_i \in \mathbf{W}$, if $\mathcal{H}(\mathbf{q}_j, \mathbf{w}_i) \leq \delta$, then $\mathbf{q}_j$ has the same component with $\mathbf{w}_i$ on at least $d-\delta$ dimensions. $\mathbf{w}_i$ has $\delta+1$ unique components, so one of $\mathbf{w}_i$'s unique components is also $\mathbf{q}_j$'s component. Hence, we have $\mathsf{ID}(\mathbf{q}_j) \cap \mathsf{ID}(\mathbf{w}_i) \neq \emptyset$.

Theorem 2 (Distinctiveness). *The protocol presented in Fig. 5 satisfies the distinctiveness defined in Definition 1.*

Proof. The distinctiveness comes from Definition 5 of unique component.

Theorem 3 (Security). *The protocol presented in Fig. 5 satisfies the security defined in Definition 1.*

Proof. Since UniqC Fmap does degenerate into two algorithms without interaction from a two-party protocol, outputs received by both parties are independent of each other's inputs. Thus, the security property is self-evident.

5.2 SAS Fmap for L_∞ Distance

We present a construction of semi-honest secure UFmap for L_∞ distance, which is denoted by SAS Fmap. We use $\mathsf{id}_{\mathbf{q}_j}$ and $\mathsf{id}_{\mathbf{w}_i}$ to represent the only element in $\mathsf{ID}(\mathbf{q}_j)$ and $\mathsf{ID}(\mathbf{w}_i)$ respectively.

As described in Sect. 2.4, for each point in input sets, SAS Fmap generates the sum, protected by DH keys of two parties, of this point's assignments in assigned coordinate systems of two parties as its identifier.

We first deal with the assignment process with spatial additive sharing (SAS), and then utilize the assignment algorithm to construct SAS Fmap.

Assignment Algorithm from SAS. SAS treats a point's assignment as the sum of its components' assignments on d dimensions, thereby converting the processing of a point in d-dimensional space into the processing of d points in 1-dimensional axes. And SAS ensures that the assignment of each component of each point is also the assignment of the $2\delta + 1$ positions centered around this component on the corresponding dimension. Our assignment algorithm is described formally in Fig. 6.

```
PARAMETERS:
  – Input size $m$.
  – Space dimension $d$.
  – Threshold $\delta$.
  – A finite group $\mathbb{G}$.
Assignment $\left(\mathbf{Q} = (\mathbf{q}_1, \cdots, \mathbf{q}_m) \in \mathbb{U}^{d\times m}\right)$:
  1. Initialize key-value pairs multiset $\mathsf{mList}_{\mathcal{S}} \leftarrow \emptyset$.
  2. For each $j \in [m]$ and each $k \in [d]$:
       Sample $\mathsf{Rand}_{j,k} \xleftarrow{R} \mathbb{G}$.
       For each $t \in [-\delta, \delta]$:
         Update $\mathsf{mList}_{\mathcal{S}} \leftarrow \mathsf{mList}_{\mathcal{S}} \cup \{(k \| (q_{j,k} + t), \mathsf{Rand}_{j,k})\}$.
         While there exists assigned $k \| (q_{j,k} + t)$ with other value in $\mathsf{mList}_{\mathcal{S}}$:
           Update values of its $2\delta + 1$ adjacent points with $\mathsf{Rand}_{j,k}$.
  3. Remove duplicates in $\mathsf{mList}_{\mathcal{S}}$.
  4. Pad $\mathsf{mList}_{\mathcal{S}}$ with dummy random elements to get $\mathsf{List}_{\mathcal{S}}$ of size $(2\delta + 1)dm$.
  5. For each $j \in [m]$:
       Set $\mathsf{Seed}_{\mathbf{q}_j,\mathcal{S}} \leftarrow \sum_{k\in[d]} \mathsf{List}_{\mathcal{S}}[k \| q_{j,k}]$.
  6. Return $\mathsf{List}_{\mathcal{S}}$ and $\left(\mathsf{Seed}_{\mathbf{q}_j,\mathcal{S}}\right)_{j\in[m]}$.
```

Fig. 6. Assignment Algorithm: $\mathsf{Assignment}(\cdot)$

UFmap Based on SAS. We assume that input sets of both parties have good distribution in a high-dimensional space. Thus, we propose a semi-honest secure UFmap based on SAS for L_∞ distance.

Intuitively, both parties first use assignment algorithm to attain their assigned coordinate systems (i.e. assigned axes of d dimensions). They encode their assigned coordinate systems into OKVS in the form of ElGamal ciphertexts and send OKVS to each other. By leveraging the homomorphism of ElGamal, both of them can obtain ciphertexts of their own points assigned in the other's coordinate system. Finally, through a masked DH subprotocol, they can securely acquire their own Fmap output. The formal description of SAS Fmap is in Fig. 7.

Definition 6 (Separated Set). *For two points* $\mathbf{q}$ *and* $\mathbf{q}'$ *in a d-dimensional space,* $\mathbf{q}$ collides *with* $\mathbf{q}'$ *on dimension k if and only if the distance between their components on dimension k is not greater than* 2δ*; otherwise,* $\mathbf{q}$ *is* separated *from* $\mathbf{q}'$ *on dimension k. A set* $\mathbf{Q}$ *is* separated*, if and only if, for each point in* $\mathbf{Q}$*, there exists a dimension such that this point is separated from anyone of other points in* $\mathbf{Q}$ *on it.*

Lemma 3 (Uniform Distribution [1]**).** *In a d-dimensional space, if points in set* $\mathbf{Q}$ *are uniformly distributed, then the probability that* $\mathbf{Q}$ *is separated is* $1 - \mathsf{negl}(d)$.

For SAS Fmap, we introduce the $\mathsf{R} \wedge \mathsf{S}$. disj. proj. assumption:

Each Sender's or Receiver's point is separated from other points in the same set on at least one dimension.

If points of inputs are uniformly distributed, then according to Lemma 3, this assumption holds in high-dimensional case with overwhelming probability, thus it is acceptable to base our construction on this assumption.

It is self-evident that the expansion rate of our protocol in Fig. 7 is 1. Now, we prove our protocol is a semi-honest secure Fmap for L_∞ distance.

Theorem 4 (Correctness). *The protocol presented in Fig. 7 satisfies the correctness defined in Definition 1 for* L_∞ *distance.*

Proof. For $\mathbf{q}_j \in \mathbf{Q}$ and $\mathbf{w}_i \in \mathbf{W}$, if $L_\infty(\mathbf{q}_j, \mathbf{w}_i) \le \delta$, then for $k \in [d]$, $|q_{j,k} - w_{i,k}| \le \delta$ always holds, thus these $q_{j,k}$ are all assigned in $\mathsf{List}_{\mathcal{R}}$. According to correctness of OKVS, $\mathcal{S}$ gets ElGamal ciphertexts of $\mathsf{List}_{\mathcal{R}}[k \| q_{j,k}]$ by decoding.

Since Assignment algorithm ensures that the $2\delta + 1$ points on dimension k centered at $w_{i,k}$ all have the same assignment, it comes that $\mathsf{List}_{\mathcal{R}}[k \| q_{j,k}] = \mathsf{List}_{\mathcal{R}}[k \| w_{i,k}]$ for $k \in [d]$. Therefore, $\mathsf{sum}_{\mathbf{q}_j}^{\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}}$ is the ElGamal ciphertext of $\mathsf{Seed}_{\mathbf{w}_i,\mathcal{R}}$. Then, it is clear that

$$\mathsf{Seed}^{\mathbf{q}_j}_{\mathsf{mk}_{\mathcal{S},j},\mathcal{R}} = \mathsf{sk}_{\mathsf{DH},\mathcal{R}} \cdot \mathsf{mask}_{\mathcal{S},j} \cdot \left(\mathsf{Seed}_{\mathbf{q}_i,\mathcal{S}} + \mathsf{Seed}_{\mathbf{w}_j,\mathcal{R}}\right)$$

$$\Longrightarrow \mathsf{id}_{\mathbf{q}_j} = \mathsf{sk}_{\mathsf{DH},\mathcal{S}} \cdot \mathsf{sk}_{\mathsf{DH},\mathcal{R}} \cdot \left(\mathsf{Seed}_{\mathbf{q}_i,\mathcal{S}} + \mathsf{Seed}_{\mathbf{w}_j,\mathcal{R}}\right)$$

Similarly, we can find that

$$\mathsf{id}_{\mathbf{w}_i} = \mathsf{sk}_{\mathsf{DH},\mathcal{R}} \cdot \mathsf{sk}_{\mathsf{DH},\mathcal{S}} \cdot \left(\mathsf{Seed}_{\mathbf{w}_j,\mathcal{R}} + \mathsf{Seed}_{\mathbf{q}_i,\mathcal{S}}\right)$$

Hence, $\mathsf{id}_{\mathbf{q}_j}$ equals $\mathsf{id}_{\mathbf{w}_i}$ when $L_\infty(\mathbf{q}_j, \mathbf{w}_i) \le \delta$ holds.

PARAMETERS:

- Sender $\mathcal{S}$ and Receiver $\mathcal{R}$.
- A finite group $\mathbb{G}$ of prime order p and $\mathbb{F}_p$.
- EC-ElGamal scheme $\mathcal{E} = \left(\mathsf{Gen}^{\mathsf{ElG}}, \mathsf{Enc}^{\mathsf{ElG}}, \mathsf{Dec}^{\mathsf{ElG}}, \oplus^{\mathsf{ElG}}\right)$ with plaintext space $\mathbb{G}$.
- An OKVS scheme $(\mathsf{Encode}, \mathsf{Decode})$ and its random value r.

INPUT OF $\mathcal{S}$: $\mathbf{Q} = (\mathbf{q}_1, \cdots, \mathbf{q}_m) \in \mathbb{U}^{d \times m}$.
INPUT OF $\mathcal{R}$: $\mathbf{W} = (\mathbf{w}_1, \cdots, \mathbf{w}_n) \in \mathbb{U}^{d \times n}$.

PROTOCOL:

Phase 1. Assignment and Summation:

1. $\mathcal{S}$ computes $\left(\mathsf{List}_{\mathcal{S}}, \left(\mathsf{Seed}_{\mathbf{q}_j,\mathcal{S}}\right)_{j\in[m]}\right) \leftarrow \mathsf{Assignment}\,(\mathbf{Q})$.
2. $\mathcal{S}$ generates $\left(\mathsf{sk}_{\mathsf{ElG},\mathcal{S}}, \mathsf{pk}_{\mathsf{ElG},\mathcal{S}}\right) \leftarrow \mathsf{Gen}^{\mathsf{ElG}}(1^{\kappa})$. $\mathcal{S}$ samples m masks $\left\{\mathsf{mask}_{\mathcal{S},j} \xleftarrow{\mathsf{R}} \mathbb{F}_p\right\}_{j\in[m]}$ and computes their inverses $\left\{\mathsf{mask}_{\mathcal{S},j}^{-1}\right\}_{j\in[m]}$.
3. $\mathcal{S}$ initializes set $\widetilde{\mathsf{List}_{\mathcal{S}}} \leftarrow \emptyset$. For each $(\mathsf{key}, \mathsf{List}_{\mathcal{S}}[\mathsf{key}]) \in \mathsf{List}_{\mathcal{S}}$, $\mathcal{S}$ updates
$$\widetilde{\mathsf{List}_{\mathcal{S}}} \leftarrow \widetilde{\mathsf{List}_{\mathcal{S}}} \cup \left\{\left(\mathsf{key}, \mathsf{Enc}^{\mathsf{ElG}}_{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}\left(\mathsf{List}_{\mathcal{S}}[\mathsf{key}]\right)\right)\right\}$$
.
4. $\mathcal{S}$ encodes $E_{\mathcal{S}} \leftarrow \mathsf{Encode}\left(\widetilde{\mathsf{List}_{\mathcal{S}}}, r\right)$, and sends $E_{\mathcal{S}}, \mathsf{pk}_{\mathsf{ElG},\mathcal{S}}$ to $\mathcal{R}$.
5. Symmetrically, $\mathcal{R}$ sends $E_{\mathcal{R}} \leftarrow \mathsf{Encode}\left(\widetilde{\mathsf{List}_{\mathcal{R}}}, r\right), \mathsf{pk}_{\mathsf{ElG},\mathcal{R}}$ to $\mathcal{S}$.
6. For each $j \in [m]$, $\mathcal{S}$ computes
$$\mathsf{sum}_{\mathbf{q}_j}^{\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}} \leftarrow \bigoplus^{\mathsf{ElG}}_{\substack{\mathsf{pk}_{\mathsf{ElG},\mathcal{R}} \\ k\in[d]}} \mathsf{Decode}\left(E_{\mathcal{R}}, k\|q_{j,k}, r\right)$$
$$\mathsf{cipher}_{\mathbf{q}_j}^{\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}} \leftarrow \mathsf{mask}_{\mathcal{S},j} \times \left(\left(\mathsf{Enc}^{\mathsf{ElG}}_{\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}}\left(\mathsf{Seed}_{\mathbf{q}_j,\mathcal{S}}\right)\right) \oplus^{\mathsf{ElG}}_{\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}} \mathsf{sum}_{\mathbf{q}_j}^{\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}}\right)$$

Phase 2. Decryption and Exchange:

7. $\mathcal{S}$ sends m ElGamal ciphertexts $\left(\mathsf{cipher}_{\mathbf{q}_j}^{\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}}\right)_{j\in[m]}$ to $\mathcal{R}$.
8. Symmetrically, $\mathcal{R}$ sends n ElGamal ciphertexts $\left(\mathsf{cipher}_{\mathbf{w}_i}^{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}\right)_{i\in[n]}$ to $\mathcal{S}$.
9. $\mathcal{S}$ samples $\mathsf{sk}_{\mathsf{DH},\mathcal{S}} \xleftarrow{\mathsf{R}} \mathbb{F}_p$. For each $i \in [n]$, $\mathcal{S}$ computes
$$\mathsf{Seed}^{\mathbf{w}_i}_{\mathsf{mk}_{\mathcal{R},i}} \leftarrow \mathsf{Dec}^{\mathsf{ElG}}_{\mathsf{sk}_{\mathsf{ElG},\mathcal{S}}}\left(\mathsf{cipher}_{\mathbf{w}_i}^{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}\right)$$
$$\mathsf{Seed}^{\mathbf{w}_i}_{\mathsf{mk}_{\mathcal{R},i},\mathcal{S}} \leftarrow \mathsf{sk}_{\mathsf{DH},\mathcal{S}} \cdot \mathsf{Seed}^{\mathbf{w}_i}_{\mathsf{mk}_{\mathcal{R},i}}$$
10. $\mathcal{S}$ sends $\left(\mathsf{Seed}^{\mathbf{w}_i}_{\mathsf{mk}_{\mathcal{R},i},\mathcal{S}}\right)_{i\in[n]}$ to $\mathcal{R}$. Symmetrically, $\mathcal{R}$ sends $\left(\mathsf{Seed}^{\mathbf{q}_j}_{\mathsf{mk}_{\mathcal{S},j},\mathcal{R}}\right)_{j\in[m]}$.
11. For each $i \in [m]$, $\mathcal{S}$ computes $\left(\mathsf{Seed}^{\mathbf{q}_j}_{\mathsf{mk}_{\mathcal{S},j},\mathcal{R},\mathcal{S}}\right) \leftarrow \mathsf{sk}_{\mathsf{DH},\mathcal{S}} \cdot \left(\mathsf{Seed}^{\mathbf{q}_j}_{\mathsf{mk}_{\mathcal{S},j},\mathcal{R}}\right)$, and unmasks $\left(\mathsf{Seed}^{\mathbf{q}_j}_{\mathsf{mk}_{\mathcal{S},j},\mathcal{R},\mathcal{S}}\right)$ to get $\mathsf{id}_{\mathbf{q}_j} \leftarrow \mathsf{mask}_{\mathcal{S},j}^{-1} \cdot \left(\mathsf{Seed}^{\mathbf{q}_j}_{\mathsf{mk}_{\mathcal{S},j},\mathcal{R},\mathcal{S}}\right)$.
12. Symmetrically, $\mathcal{R}$ gets $(\mathsf{id}_{\mathbf{w}_i})_{i\in[n]}$.

Fig. 7. UFmap Protocol for L_∞ Distance Without Expansion: $\Pi^{L_\infty}_{\mathsf{SAS\ Fmap}}$

Theorem 5 (Distinctiveness). *The protocol presented in Fig. 7 satisfies the distinctiveness defined in Definition 1.*

Proof. First consider the side of $\mathcal{S}$. For $j \in [m]$, let us assume that $\mathbf{q}_j$ is separated from the others in $\mathbf{W}$ on dimension k_j. With reference to proof of Theorem 4, we have

$$\mathsf{id}_{\mathbf{q}_j} \triangleq \mathsf{sk}_{\mathsf{DH},\mathcal{S}} \cdot \mathsf{sk}_{\mathsf{DH},\mathcal{R}} \cdot \left(\mathsf{List}_{\mathcal{S}}\left[k_j \| q_{j,k_j}\right] + \Delta_j\right)$$

Under $\mathsf{R} \wedge \mathsf{S}$. disj. proj. assumption, in assignment algorithm, the $2\delta + 1$ points centered at q_{j,k_j} on dimension k_j do not cover any other assigned points nor be covered by any other assigned points. Thus, $\mathsf{List}_{\mathcal{S}}\left[k_j \| q_{j,k_j}\right]$ is a uniformly random value independent of any other assignments. So, the probability that $\mathsf{id}_{\mathbf{q}_j}$ equals some $\mathsf{id}_{\mathbf{q}_{j'}}$ is $\frac{m-1}{|\mathbb{G}|}$. Hence, it holds that

$$\Pr\left[\exists\ j, j' \in [m], s.t. (j \neq j') \wedge \left(\mathsf{ID}\left(\mathbf{q}_j\right) \cap \mathsf{ID}\left(\mathbf{q}_{j'}\right) \neq \emptyset\right)\right] \leq \frac{m^2}{|\mathbb{G}|} = \mathsf{negl}\left(\kappa\right)$$

Symmetrically, the same discussion for $\mathcal{R}$ will complete the proof.

Theorem 6 (Randomness). *In the protocol presented in Fig. 7, $\left(\mathsf{id}_{\mathbf{q}_j}\right)_{j\in[m]}$ is computationally indistinguishable from $\mathsf{R}_{\mathsf{id}} \overset{\mathsf{R}}{\longleftarrow} \mathbb{G}^m$, and $\left(\mathsf{id}_{\mathbf{w}_i}\right)_{i\in[n]}$ is computationally indistinguishable from $\mathsf{R}_{\mathsf{id}} \overset{\mathsf{R}}{\longleftarrow} \mathbb{G}^n$, if the DDH assumption holds.*

Proof. First consider the side of $\mathcal{S}$. With proof of Theorem 4, we have

$$\mathsf{id}_{\mathbf{q}_j} = \mathsf{sk}_{\mathsf{DH},\mathcal{S}} \cdot \left(\mathsf{sk}_{\mathsf{DH},\mathcal{R}} \cdot \mathsf{Seed}_{\mathbf{q}_j,\mathcal{S}}\right) + \mathsf{sk}_{\mathsf{DH},\mathcal{S}} \cdot \mathsf{sk}_{\mathsf{DH},\mathcal{R}} \cdot \mathsf{Seed}_{\mathbf{q}_j,\mathcal{R}}$$

According to DDH assumption, $\left(\mathsf{sk}_{\mathsf{DH},\mathcal{R}} \cdot \mathsf{Seed}_{\mathbf{q}_j,\mathcal{S}}\right)_{j\in[m]}$ is computationally indistinguishable from uniformly random vector in $\mathbb{G}^m$.

Since the assignment of $\mathcal{S}$'s coordinate system and that of $\mathcal{R}$'s coordinate system are independent, $\mathsf{Seed}_{\mathbf{q}_j,\mathcal{R}}$ is independent of $\mathsf{Seed}_{\mathbf{q}_j,\mathcal{S}}$. In conclusion, $\left(\mathsf{id}_{\mathbf{q}_j}\right)_{j\in[m]}$ is computationally indistinguishable from $\mathsf{R}_{\mathsf{id}} \overset{\mathsf{R}}{\longleftarrow} \mathbb{G}^m$.

Symmetrically, the same discussion for $\mathcal{R}$ will complete the proof.

Theorem 7 (Security). *The protocol presented in Fig. 7 satisfies the security defined in Definition 1 if the DDH assumption holds.*

Proof. First consider the side of $\mathcal{S}$. We exhibit simulator $\mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}\left(\mathbf{Q}, \mathsf{id}\left(\mathbf{Q}\right)_{\mathbf{W}}\right)$ for simulating corrupt $\mathcal{S}$ where $\mathsf{id}\left(\mathbf{Q}\right)_{\mathbf{W}}$ is the output of $\mathcal{S}$ holding $\mathbf{Q}$ who invokes the protocol presented in Fig. 7 with $\mathcal{R}$ holding $\mathbf{W}$. And we argue the indistinguishability of the produced transcript from the real execution.

$\mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}$ simulating the view of corrupt semi-honest Sender executes as follows:

1. $\mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}$ generates $\left(\overline{\mathsf{sk}_{\mathsf{ElG},\mathcal{S}}}, \overline{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}\right) \leftarrow \mathsf{Gen}^{\mathsf{ElG}}\left(1^\kappa\right)$, samples $\left\{\overline{\mathsf{mask}_{\mathcal{S},j}} \overset{\mathsf{R}}{\longleftarrow} \mathbb{F}_p\right\}_{j\in[m]}$, computes $\left\{\overline{\mathsf{mask}_{\mathcal{S},j}}^{-1}\right\}_{j\in[m]}$, and appends them to the view.

2. $\mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}$ encodes OKVS $\overline{E_{\mathcal{R}}}$ with $(2\delta+1)dn$ dummy key-value pairs, generates $\left(\overline{\mathsf{sk}_{\mathsf{ElG},\mathcal{R}}}, \overline{\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}}\right) \leftarrow \mathsf{Gen}^{\mathsf{ElG}}(1^{\kappa})$, and appends $\overline{E_{\mathcal{R}}}, \overline{\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}}$ to the view.
3. $\mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}$ samples $\left(\overline{\mathsf{Seed}_i} \xleftarrow{R} \mathbb{G}\right)_{i\in[n]}$, computes $\left(\overline{\mathsf{cipher}_i^{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}} \leftarrow \mathsf{Enc}^{\mathsf{ElG}}_{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}(\overline{\mathsf{Seed}_i})\right)_{i\in[n]}$, and appends $\left(\overline{\mathsf{cipher}_i^{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}}\right)_{i\in[n]}$ to the view.
4. $\mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}$ samples $\overline{\mathsf{sk}_{\mathsf{DH},\mathcal{S}}} \xleftarrow{R} \mathbb{F}_p$ and appends it to the view.
5. $\mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}$ computes $\left(\overline{\mathsf{Seed}^{\mathbf{q}_j}_{\mathsf{mk}_{\mathcal{S},j},\mathcal{R}}} \leftarrow \overline{\mathsf{mask}_{\mathcal{S},j}} \cdot \overline{\mathsf{sk}_{\mathsf{DH},\mathcal{S}}}^{-1} \cdot \mathsf{id}\left(\mathbf{q}_j\right)_{\mathbf{W}}\right)_{j\in[m]}$ and appends them to the view.

Now we show that the view output by $\mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}$ is indistinguishable from the real one via a hybrid argument. We define four hybrid transcripts T_0, T_1, T_2, T_3, where T_0 is the real view of $\mathcal{S}$, and T_3 is the output of $\mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}$.

- $\mathbf{Hyb}_0$**.** This hybrid is the real interaction described in Fig. 7. Let T_0 denote $\mathcal{S}$'s view in the real protocol.
- $\mathbf{Hyb}_1$**.** Let T_1 be the same as T_0, except that OKVS $E_{\mathcal{R}}$ and $\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}$ are replaced by $\overline{E_{\mathcal{R}}}$ and $\overline{\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}}$. The values for encoding $E_{\mathcal{R}}$ are $(2\delta+1)dn$ ciphertexts encrypted with $\mathsf{pk}_{\mathsf{ElG},\mathcal{R}}$, which are computationally indistinguishable from uniformly random ElGamal ciphertexts by DDH assumption. Combining the obliviousness of OKVS, $E_{\mathcal{R}}$ and $\overline{E_{\mathcal{R}}}$ are computationally indistinguishable. Hence, we have $T_1 \overset{c}{\approx} T_0$.
- $\mathbf{Hyb}_2$**.** Let T_2 be the same as T_1, except that $\left(\mathsf{sk}_{\mathsf{ElG},\mathcal{S}}, \mathsf{pk}_{\mathsf{ElG},\mathcal{S}}\right)$ and $\left(\mathsf{cipher}_{\mathbf{w}_i}^{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}\right)_{i\in[n]}$ are replaced by $\left(\overline{\mathsf{sk}_{\mathsf{ElG},\mathcal{S}}}, \overline{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}\right)$ and $\left(\overline{\mathsf{cipher}_i^{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}}\right)_{i\in[n]}$. Since each $\mathsf{Seed}^{\mathbf{w}_i}_{\mathsf{mk}_{\mathcal{R},i}}$ is masked by uniformly random $\mathsf{mask}_{\mathcal{R},i}$ from $\mathcal{R}$, $\left(\mathsf{Seed}^{\mathbf{w}_i}_{\mathsf{mk}_{\mathcal{R},i}}\right)_{i\in[n]}$ are statistically indistinguishable from $\left(\overline{\mathsf{Seed}_i} \xleftarrow{R} \mathbb{G}\right)_{i\in[n]}$. Therefore, $\left(\mathsf{cipher}_{\mathbf{w}_i}^{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}\right)_{i\in[n]}$ and $\left(\overline{\mathsf{cipher}_i^{\mathsf{pk}_{\mathsf{ElG},\mathcal{S}}}}\right)_{i\in[n]}$ are statistically indistinguishable, which means $T_2 \overset{s}{\approx} T_1$.
- $\mathbf{Hyb}_3$**.** Let T_3 be the same as T_2, except that $\{\mathsf{mask}_{\mathcal{S},j}\}_{j\in[m]}$, $\left\{{\mathsf{mask}_{\mathcal{S},j}}^{-1}\right\}_{j\in[m]}$, $\mathsf{sk}_{\mathsf{DH},\mathcal{S}}$, and $\left(\mathsf{Seed}^{\mathbf{q}_j}_{\mathsf{mk}_{\mathcal{S},j},\mathcal{R}}\right)_{j\in[m]}$ are replaced by $\left\{\overline{\mathsf{mask}_{\mathcal{S},j}}\right\}_{j\in[m]}$, $\left\{\overline{\mathsf{mask}_{\mathcal{S},j}}^{-1}\right\}_{j\in[m]}$, $\overline{\mathsf{sk}_{\mathsf{DH},\mathcal{S}}}$, and $\left(\overline{\mathsf{Seed}^{\mathbf{q}_j}_{\mathsf{mk}_{\mathcal{S},j},\mathcal{R}}}\right)_{j\in[m]}$. It is clear that masks $\{\mathsf{mask}_{\mathcal{S},j}\}_{j\in[m]}$ and $\left\{\overline{\mathsf{mask}_{\mathcal{S},j}}\right\}_{j\in[m]}$ are distributed identically; $\mathsf{sk}_{\mathsf{DH},\mathcal{S}}$ and $\overline{\mathsf{sk}_{\mathsf{DH},\mathcal{S}}}$ are distributed identically. Hence, $\left(\mathsf{Seed}^{\mathbf{q}_j}_{\mathsf{mk}_{\mathcal{S},j},\mathcal{R}}\right)_{j\in[m]}$ and $\left(\overline{\mathsf{Seed}^{\mathbf{q}_j}_{\mathsf{mk}_{\mathcal{S},j},\mathcal{R}}}\right)_{j\in[m]}$ are statistically indistinguishable. Thus $T_3 \overset{s}{\approx} T_2$ holds.

From the argument above, it holds that

$$\mathsf{view}_{\mathcal{R}}^{\Pi_{\mathsf{SAS}\ \mathsf{Fmap}}}(\kappa, \lambda; \mathbf{Q}, \mathbf{W}) \overset{c}{\approx} \mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}\left(\mathbf{Q}, \mathsf{id}\left(\mathbf{Q}\right)_{\mathbf{W}}\right)$$

In addition, according to Theorem 6, we have

$$\mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}\left(\mathbf{Q}, \mathsf{id}\left(\mathbf{Q}\right)_{\mathbf{W}}\right) \overset{c}{\approx} \mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}\left(\mathbf{Q}, \mathsf{R}_{\mathsf{id}} \overset{\mathsf{R}}{\leftarrow} \mathbb{G}^{m}\right)$$

Therefore, it comes that

$$\mathsf{view}^{\Pi_{\mathsf{SAS}}\ \mathsf{Fmap}}_{\mathcal{R}}\left(\kappa, \lambda; \mathbf{Q}, \mathbf{W}\right) \overset{c}{\approx} \mathsf{Sim}^{\mathcal{S}}_{\mathsf{Fmap}}\left(\mathbf{Q}, \mathsf{R}_{\mathsf{id}} \overset{\mathsf{R}}{\leftarrow} \mathbb{G}^{m}\right) \overset{c}{\approx} \mathsf{view}^{\Pi_{\mathsf{SAS}}\ \mathsf{Fmap}}_{\mathcal{R}}\left(\kappa, \lambda; \mathbf{Q}, \mathbf{W}'\right)$$

Symmetrically, the same discussion for $\mathcal{R}$ will complete the proof.

Remark 3 (Complexity). The protocol presented in Fig. 7 has communication complexity $\mathcal{O}\left(\left((2\delta+1)d\kappa + 2\kappa + \kappa\right)(m+n)\right)$, computation complexity $\mathcal{O}\left((2\delta+1)dm + n\right)$ for $\mathcal{S}$, and computation complexity $\mathcal{O}\left((2\delta+1)dn + m\right)$ for $\mathcal{R}$, if the OKVS has a constant rate, linear encoding time, and constant decoding time.

6 Multi-query Fuzzy RPMT Based on sUFmap

In this section, we provide the ideal functionality for mqFRPMT and present mqFRPMT protocols for L_∞ distance and for $L_{\mathsf{p}\in[1,\infty)}$ distance respectively, using sUFmap for L_∞ distance.

6.1 Definition of mqFRPMT

mqFRPMT is the fuzzy version of mqRPMT, and we define the ideal functionality for mqFRPMT in Fig. 8. Combining with OT, mqFRPMT can directly yield FPSI, FPSI-card, and LFPSI.

PARAMETERS: Sender $\mathcal{S}$, Receiver $\mathcal{R}$; Set size m, n; Dimension d; Distance function $\mathsf{dist}(\cdot,\cdot)$; Distance threshold δ.

FUNCTIONALITY:

- Wait an input $\mathbf{Q} \in \mathbb{U}^{d\times m}$ from $\mathcal{S}$.
- Wait an input $\mathbf{W} \in \mathbb{U}^{d\times n}$ from $\mathcal{R}$.
- Return $\mathbf{e} = (e_1, \cdots, e_m) \in \{0,1\}^m$ to $\mathcal{R}$, where $e_j = 1$ if and only if there exists $\mathbf{w}_i \in \mathbf{W}$ such that $\mathsf{dist}\left(\mathbf{q}_j, \mathbf{w}_i\right) \le \delta$.

Fig. 8. Ideal Functionality for Multi-Query Fuzzy RPMT $\mathcal{F}_{\mathsf{mqFRPMT}}$

6.2 mqFRPMT for L_∞ Distance from sUFmap

The high-level idea of sUFmap-based mqFRPMT is as described in Sect. 2.5. In mqFRPMT for L_∞ distance, we instantiate fuzzy matching with an idea similar to [1]. We give the detailed mqFRPMT protocol for L_∞ distance in Fig. 9.

We provide the proofs of correctness and security in the full version.

Parameters:

- Sender $\mathcal{S}$ and Receiver $\mathcal{R}$.
- Space dimension d and threshold δ.
- An AHE scheme $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \oplus)$.
- An OKVS scheme $(\mathsf{Encode}, \mathsf{Decode})$ and its random value r.

Input of $\mathcal{S}$: $\mathbf{Q} = (\mathbf{q}_1, \cdots, \mathbf{q}_m) \in \mathbb{U}^{d \times m}$.
Input of $\mathcal{R}$: $\mathbf{W} = (\mathbf{w}_1, \cdots, \mathbf{w}_n) \in \mathbb{U}^{d \times n}$.

Protocol:

1. $\mathcal{S}$ and $\mathcal{R}$ invoke $\Pi^{L_\infty}_{\mathsf{sUFmap}}$: $\mathcal{S}$ acts as Sender with input $\mathbf{Q}$ and $\mathcal{R}$ acts as Receiver with input $\mathbf{W}$. $\mathcal{S}$ receives $\mathsf{ID}(\mathbf{Q})$ and $\mathcal{R}$ receives $\mathsf{ID}(\mathbf{W})$.
2. $\mathcal{R}$ generates $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\kappa)$. $\mathcal{R}$ initializes set $\mathsf{List} \leftarrow \emptyset$.
3. For each $i \in [n]$, each $k \in [d]$, and each $t \in [-\delta, \delta]$, $\mathcal{R}$ update
$$\mathsf{List} \leftarrow \mathsf{List} \cup \{(\mathsf{id}_{\mathbf{w}_i} \| k \| (w_{i,k} + t), \mathsf{Enc}_{\mathsf{pk}}(0))\}_{\mathsf{id}_{\mathbf{w}_i} \in \mathsf{ID}(\mathbf{w}_i)}$$
4. $\mathcal{R}$ encodes $E \leftarrow \mathsf{Encode}(\mathsf{List}, r)$, and sends E, pk to $\mathcal{S}$.
5. For each $j \in [m]$, $\mathcal{S}$ samples $\mathsf{mask}_j \xleftarrow{\mathsf{R}} \mathcal{P}$ and computes
$$\mathsf{cipher}_j \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathsf{mask}_j) \oplus_{\mathsf{pk}} \left(\bigoplus_{k \in [d]}{}_{\mathsf{pk}} \mathsf{Decode}\left(E, \mathsf{id}_{\mathbf{q}_j} \| k \| q_{j,k}, r\right) \right)$$
Then, $\mathcal{S}$ sends $\left(\mathsf{cipher}_j\right)_{j \in [m]}$ to $\mathcal{R}$.
6. $\mathcal{R}$ computes $\left(v_j \leftarrow \mathsf{Dec}_{\mathsf{sk}}\left(\mathsf{cipher}_j\right)\right)_{j \in [m]}$.
7. For each $j \in [m]$, $\mathcal{S}$ and $\mathcal{R}$ invoke $\mathcal{F}_{\mathsf{PEqT}}$: $\mathcal{S}$ acts as Sender with input mask_j and $\mathcal{R}$ acts as Receiver with input v_j. $\mathcal{S}$ receives nothing and $\mathcal{R}$ receives e_j.
8. $\mathcal{R}$ learns $\mathbf{e} = (e_1, \cdots, e_m)$.

Fig. 9. mqFRPMT for L_∞ from sUFmap: $\Pi^{L_\infty}_{\mathsf{mqFRPMT}}$

Theorem 8 (Correctness). *The protocol presented in Fig. 9 realizes the functionality $\mathcal{F}_{\mathsf{mqFRPMT}}$ defined in Fig. 8 for L_∞ distance correctly.*

Theorem 9 (Security). *The protocol presented in Fig. 9 realizes the functionality $\mathcal{F}_{\mathsf{mqFRPMT}}$ defined in Fig. 8 for L_∞ distance against semi-honest adversaries in the $\mathcal{F}_{\mathsf{PEqT}}$-hybrid model if $\mathcal{E}$ satisfies IND-CPA security.*

Remark 4 (Complexity). The protocol presented in Fig. 9 has communication complexity $\mathcal{O}(((2\delta + 1)d\kappa + 2\kappa + \kappa)(m + n))$, computation complexity $\mathcal{O}((2\delta + 1)dm + n)$ for $\mathcal{S}$, and computation complexity $\mathcal{O}((2\delta + 1)dn + m)$ for $\mathcal{R}$, if the OKVS has a constant rate, linear encoding time, and constant decoding time; the UFmap is SAS Fmap in Fig. 7.

6.3 mqFRPMT for $L_{\mathsf{p}} \in [1, \infty)$ Distance from sUFmap

The construction of mqFRPMT for $L_{\mathsf{p} \in [1,\infty)}$ distance is similar to $\Pi^{L_\infty}_{\mathsf{mqFRPMT}}$. For computing $L_{\mathsf{p} \in [1,\infty)}$ distance, in OKVS encoding, AHE ciphertexts of $|t|^{\mathsf{p}}$

instead of 0 are used as values of $\mathsf{id}_{\mathbf{w}_i} \| (w_{i,k} + t)$. Therefore, Sender can compute AHE ciphertexts of the p power of distances and mask them. With IFmat protocol, Receiver can complete the secure comparison between masked p power of distances and masked δ^{p} with Sender to learn final result of mqFRPMT. We give the detailed protocol and relevant proofs in the full version.

7 FPSI Protocols

7.1 Generic Construction of FPSI from Fmap

Fmap generates the same identifier for Sender's point and Receiver's point that are close to each other, and then Sender and Receiver can use OKVS and fuzzy matching to further filter the point pairs implied by these identifiers to obtain the FPSI output. This is a generic approach to constructing FPSI from Fmap, indicating the adaptability of Fmap for various distance functions.

FPSI for Distances with Translation Invariance. As a specific example, let us now focus on constructing FPSI from Fmap for those distance functions having the translation invariance property.

Definition 7 (Translation Invariance). *A distance function* $\mathsf{dist}(\cdot,\cdot)$ *on* $\mathbb{U}^d \times \mathbb{U}^d$ *has* translation invariance *property if and only if, for any two point* $\mathbf{q}, \mathbf{w} \in \mathbb{U}^d$ *and any vector* $\mathbf{v} \in \mathbb{U}^d$, *it holds that*

$$\mathsf{dist}(\mathbf{q}, \mathbf{w}) = \mathsf{dist}(\mathbf{q} + \mathbf{v}, \mathbf{w} + \mathbf{v})$$

It is not difficult to see that Hamming and $L_{\mathsf{p}\in[1,\infty]}$ distances both have translation invariance property. We provide the detailed generic construction from Fmap to FPSI for distance with translation invariance in Fig. 10. Thus, this generic construction is a powerful tool for FPSI for Hamming and $L_{\mathsf{p}\in[1,\infty]}$ distances. Specifically, we can instantiate the Fmap and fuzzy matching in the construction in Fig. 10 with UniqC Fmap and trivial fuzzy matching for Hamming distance in Sect. 3.3 to obtain an FPSI for Hamming distance.

We provide the proofs of correctness and security in the full version.

Theorem 10 (Correctness). *The protocol presented in Fig. 10 realizes the functionality* $\mathcal{F}_{\mathsf{FPSI}}$ *defined in Fig. 1 for distance with translation invariance correctly.*

Theorem 11 (Security). *The protocol presented in Fig. 10 realizes the functionality* $\mathcal{F}_{\mathsf{FPSI}}$ *defined in Fig. 1 for distance with translation invariance against semi-honest adversaries in the* $(\mathcal{F}_{\mathsf{ssFMatch}}, \mathcal{F}_{\mathsf{PEqT}}, \mathcal{F}_{\mathsf{OT}})$*-hybrid model, if* $\mathcal{E}$ *satisfies IND-CPA security.*

Remark 5 (Costs Analysis). The communication cost of protocol presented in Fig. 10 consists of: communication cost of Fmap, sending OKVS from $n \cdot \mathsf{rate}_{\mathcal{R}}$

PARAMETERS:

- Sender $\mathcal{S}$ and Receiver $\mathcal{R}$.
- Space dimension d, distance function $\mathsf{dist}(\cdot,\cdot)$ having translation invariance, and threshold δ.
- An AHE scheme $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \oplus)$.
- An OKVS scheme $(\mathsf{Encode}, \mathsf{Decode})$ and its random value r.

INPUT OF $\mathcal{S}$: $\mathbf{Q} = (\mathbf{q}_1, \cdots, \mathbf{q}_m) \in \mathbb{U}^{d\times m}$.
INPUT OF $\mathcal{R}$: $\mathbf{W} = (\mathbf{w}_1, \cdots, \mathbf{w}_n) \in \mathbb{U}^{d\times n}$.

PROTOCOL:

0. $\mathcal{S}$ permutates $\mathbf{Q}$ randomly.
1. $\mathcal{S}$ and $\mathcal{R}$ invoke $\Pi^{\mathsf{dist}(\cdot,\cdot)}_{\mathsf{Fmap}}$: $\mathcal{S}$ acts as Sender with input $\mathbf{Q}$ and $\mathcal{R}$ acts as Receiver with input $\mathbf{W}$. $\mathcal{S}$ receives $\mathsf{ID}(\mathbf{Q})$ and permutates each $\mathsf{ID}(\mathbf{q}_j)$ randomly; $\mathcal{R}$ receives $\mathsf{ID}(\mathbf{W})$.
2. $\mathcal{R}$ generates $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\kappa)$. $\mathcal{R}$ initializes set $\mathsf{List} \leftarrow \emptyset$.
3. For each $i \in [n]$, $\mathcal{R}$ update
$$\mathsf{List} \leftarrow \mathsf{List} \cup \left\{ \left(\mathsf{id}_{\mathbf{w}_i}, \left(\mathsf{Enc}_{\mathsf{pk}}(w_{i,1}), \cdots, \mathsf{Enc}_{\mathsf{pk}}(w_{i,d}) \right) \right) \right\}_{\mathsf{id}_{\mathbf{w}_i} \in \mathsf{ID}(\mathbf{w}_i)}$$
4. $\mathcal{R}$ encodes $E \leftarrow \mathsf{Encode}(\mathsf{List}, r)$, and sends E, pk to $\mathcal{S}$.
5. For each $j \in [m]$ and each $\ell \in [\mathsf{rate}_{\mathcal{S}}]$, $\mathcal{S}$ samples $\mathbf{mask}_{j,\ell} = (\mathsf{mask}_{j,\ell,k})_{k\in[d]} \xleftarrow{\mathsf{R}} \mathbb{U}^d$ and computes
$$\mathbf{cipher}_{j,\ell} \leftarrow \left(\mathsf{Enc}_{\mathsf{pk}}(\mathsf{mask}_{j,\ell,k}) \oplus_{\mathsf{pk}} \mathsf{Decode}\left(E, \mathsf{id}_{\mathbf{q}_j,\ell}, r\right)_k \right)_{k\in[d]}$$
$$\mathbf{u}_{j,\ell} \leftarrow \mathbf{mask}_{j,\ell} + \mathbf{q}_j$$
Then, $\mathcal{S}$ sends $\left(\mathbf{cipher}_{j,\ell}\right)_{j\in[m],\ell\in[\mathsf{rate}_{\mathcal{S}}]}$ to $\mathcal{R}$.
6. $\mathcal{R}$ computes $\left(\mathbf{v}_{j,\ell} \leftarrow \mathsf{Dec}_{\mathsf{sk}}\left(\mathbf{cipher}_{j,\ell}\right)\right)_{j\in[m],\ell\in[\mathsf{rate}_{\mathcal{S}}]}$.
7. For each $j \in [m]$ and each $\ell \in [\mathsf{rate}_{\mathcal{S}}]$, $\mathcal{S}$ and $\mathcal{R}$ invoke $\mathcal{F}^{\mathsf{dist}(\cdot,\cdot)}_{\mathsf{ssFMatch}}$: $\mathcal{S}$ acts as Sender with input $\mathbf{u}_{j,\ell}$ and $\mathcal{R}$ acts as Receiver with input $\mathbf{v}_{j,\ell}$. $\mathcal{S}$ receives $e_{\mathcal{S},j,\ell}$ and $\mathcal{R}$ receives $e_{\mathcal{R},j,\ell}$.
8. For each $j \in [m]$, $\mathcal{S}$ and $\mathcal{R}$ invoke $\mathcal{F}_{\mathsf{PEqT}}$: $\mathcal{S}$ acts as Sender with input $(e_{\mathcal{S},j,\ell})_{\ell\in[\mathsf{rate}_{\mathcal{S}}]}$ and $\mathcal{R}$ acts as Receiver with input $(e_{\mathcal{R},j,\ell})_{\ell\in[\mathsf{rate}_{\mathcal{S}}]}$. $\mathcal{S}$ receives nothing and $\mathcal{R}$ receives $1 - e_j$. $\mathcal{R}$ learns $\mathbf{e} = (e_1, \cdots, e_m)$.
9. For each $j \in [m]$, $\mathcal{S}$ and $\mathcal{R}$ invoke $\mathcal{F}_{\mathsf{OT}}$: $\mathcal{S}$ acts as Sender with input $(m_0 = \perp, m_1 = \mathbf{q}_j)$ and $\mathcal{R}$ acts as Receiver with input e_j. $\mathcal{S}$ receives nothing and $\mathcal{R}$ receives m_{e_j}.
10. $\mathcal{R}$ learns $\mathbf{I}_{\mathsf{fuzzy}} \leftarrow \left\{ m_{e_j} | (j \in [m]) \wedge (e_j = 1) \right\}$ as output.

Fig. 10. FPSI for $\mathsf{dist}(\cdot,\cdot)$ with translation invariance from Fmap: $\Pi^{\mathsf{dist}(\cdot,\cdot)}_{\mathsf{FPSI}}$

pairs, sending $m \cdot \mathsf{rate}_{\mathcal{S}}$ masked ciphers of points, communication cost of $m \cdot \mathsf{rate}_{\mathcal{S}}$ fuzzy matching, and communication cost of m OTs.

For $\mathcal{S}$, the computation cost of this protocol consists of: computation cost of Fmap as Sender, $m \cdot \mathsf{rate}_{\mathcal{S}}$ decoding of OKVS, $m \cdot \mathsf{rate}_{\mathcal{S}}$ homomorphic masking of points, computation cost of $m \cdot \mathsf{rate}_{\mathcal{S}}$ fuzzy matching as Sender, and computation cost of m OTs as Sender.

For $\mathcal{R}$, the computation cost of this protocol consists of: computation cost of Fmap as Receiver, $n \cdot \mathsf{rate}_{\mathcal{R}}$ encryptions of points, encoding of OKVS with $n \cdot \mathsf{rate}_{\mathcal{R}}$ pairs, $m \cdot \mathsf{rate}_{\mathcal{S}}$ decryptions of points, computation cost of $m \cdot \mathsf{rate}_{\mathcal{S}}$ fuzzy matching as Receiver, and computation cost of m OTs as Receiver.

FPSI for Functions with Invariance. Note that our construction is not limited to distance functions with translation invariance, such as Hamming distance. For any function with some invariance, we can obtain FPSI from Fmap using a similar construction.

For example, a generic construction for function with rotation invariance, such as cosine similarity, can be proposed via simply replacing additive masks and AHE by rotational masks and homomorphic encryption allowing rotation on ciphertexts.

7.2 FPSI(-Variants) from mqFRPMT

As shown in Sect. 2.6, mqFRPMT can be used as a central building block to construct FPSI and its various variants, including LFPSI, FPSI-card, and the special FPSI-SP. For lack of space, we put the detailed method and proofs in the full version.

In Sect. 6.2 and Sect. 6.3, we present mqFRPMT protocols for L_∞ and $L_{\mathsf{p}\in[1,\infty)}$ distances respectively. Based on them, we can easily obtain FPSI for $L_{\mathsf{p}\in[1,\infty]}$ distance.

8 Implementation

We provide experimental details and specific data for FPSI, and compare our performance with previous works. We also conduct experiments in unbalanced setting and the data can be found in the full version.

8.1 Implementation Details

Environment. We run the experiments on a single machine with 2.00 GHz Intel Xeon Gold 6330 CPU and 256 GB RAM. We measure the time of online phase in a local network setting with network latency of 0.02 ms and bandwidth of 10 Gbps.

Instantiations. We choose the computational security parameter $\kappa = 128$ and the statistical security parameter $\lambda = 40$. Our protocols are written in C++ and we use the following instantiations in our implementation.

– OKVS: We use RB-OKVS in [2].

- OT: We use OT implementation in libOTe[11].
- Goldwasser-Micali: We use GMP[12] to implement Goldwasser-Micali cryptosystem with key size of 2048-bit as AHE in our FPSI for Hamming distance.
- Paillier: We use the implementation of Paillier in Intel Paillier Cryptosystem Library[13] with key size of 2048-bit as AHE in our FPSI for $L_{\mathsf{p}\in[1,\infty]}$ distance.
- Others: We use Curve25519 in cryptoTools[14] as the underlying group $\mathbb{G}$ for SAS Fmap. We adopt Coproto[15] to realize network communication.

8.2 Performance

FPSI for Hamming Distance. We compare our FPSI form UniqC Fmap for Hamming distance in Sect. 7.1 with the near-linear protocol by Chongchitmate et al., which is the only one overcomes the $m \cdot n$ blowup in complexity among prior works for Hamming distance [8]. Unfortunately, we do not have their code, thus we use their experimental results directly from their paper [8] and run our code with the same parameters.

The comparison is shown in Table 3. It can be observed that as m and n increase from 256 to 4096, the communication and computation costs of our protocol both scale linearly, and our protocol performs better than [8] in all cases. Note that our protocol achieves a 4.6× reduction in communication cost, which is independent of the running environment.

Table 3. Communication cost and running time of FPSI for Hamming distance, where input set sizes $m = n \in \{256, 1024, 4096\}$, universe $\mathbb{U} = \mathbb{F}_2$, dimension $d = 128$, and threshold $\delta = 4$. UniqC Fmap packs $P = 24$ bits as one super-component.

$m = n$	Protocol	Comm. (MB)	Time (s)
256	[8]	465.68	38.7
	Ours	**91.889**	**5.18**
1024	[8]	1779.3	147.85
	Ours	**367.53**	**19.428**
4096	[8]	6870	569.9
	Ours	**1470**	**76.00**

FPSI for $L_{\mathsf{p}\in[1,\infty]}$ Distance. We compare our FPSI from SAS Fmap in Sect. 7.2 with the state-of-the-art protocols in [1] including FPSI in low-dimensional (denoted by [1]L) and high-dimensional (denoted by [1]H) space.

[11] https://github.com/osu-crypto/libOTe.git.
[12] https://gmplib.org/.
[13] https://github.com/intel/pailliercryptolib.git.
[14] https://github.com/ladnir/cryptoTools.git.
[15] https://github.com/Visa-Research/coproto.git.

We report the performances for input sizes $m = n \in \{2^4, 2^8, 2^{12}, 2^{16}\}$, dimension $d \in \{2, 6, 10\}$, and threshold $\delta \in \{10, 30\}$. Since [1]H needs more than 10^4 seconds when $n \geq 2^{12}$, we omit these data in our tables.

FPSI for $L_{\mathsf{p}\in\{1,2\}}$ Distance. Since there is no implementation of [1]H for $L_{\mathsf{p}\in[1,\infty)}$ distance, we estimate its costs with the hyper-parameter $\rho = 0.5$ for L_1 distance and $\rho = 0.365$ for L_2 distance as reported in [1]. For comparison, we assume that the costs of [1]H only consist of OKVS encoding and sending, and estimate the encoding to take 800 machine cycles per pair, which is the best performance of our machine. In short, we report a conservative estimates of [1]H for $L_{\mathsf{p}\in\{1,2\}}$ distance in our table. Table 4 shows that, for L_1 and L_2 distance, our protocol achieves a $28 - 166\times$ speedup and reduces communication cost by a factor of $6 - 40\times$ when $d \geq 6$.

Table 4. Communication cost (MB) and running time (s) of FPSI for $L_{\mathsf{p}\in\{1,2\}}$ distance.

$m = n$	Protocol	(d, δ)											
		(2,10)		(6,10)		(10,10)		(2,30)		(6,30)		(10,30)	
		Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time
		L_1 Distance											
2^4	[1]H	4.512	31.64	13.54	94.93	22.56	158.2	13.11	87.96	39.32	263.9	65.53	439.8
	[1]L	**0.178**	0.692	8.257	24.10	220.0	677.0	**0.532**	1.888	24.77	73.69	660.0	2042
	Ours	0.469	**0.374**	**1.371**	**0.840**	**2.274**	**1.261**	1.330	**0.742**	**3.951**	**1.884**	**6.570**	**2.636**
2^8	[1]H	290.7	2084	872.1	6253	1453	10422	844.5	5714	2533	17140	4222	28567
	[1]L	**2.854**	9.296	132.1	409.7	3520	11034	**8.510**	25.70	396.3	1225	$> 10^4$	$> 10^4$
	Ours	7.502	**4.057**	**21.84**	**9.570**	**36.38**	**15.23**	21.28	**8.433**	**63.21**	**22.81**	**105.2**	**37.37**
2^{12}	[1]L	**45.66**	148.2	2113	6630	$> 10^4$	$> 10^5$	**136.2**	433.3	> 6000	$> 10^4$	$> 10^5$	$> 10^5$
	Ours	120.0	**56.92**	**351.0**	**155.0**	**589.2**	**260.2**	340.3	**130.2**	**1024**	**395.1**	**1703**	**650.4**
2^{16}	[1]L	**730.5**	2480	$> 10^4$	$> 10^5$	$> 10^5$	$> 10^6$	**2179**	7008	$> 10^4$	$> 10^5$	$> 10^6$	$> 10^6$
	Ours	1919	**966.3**	**5685**	**2736**	**9427**	**4359**	5513	**2238**	**16382**	**6416**	**27253**	**10800**
		L_2 Distance											
2^4	[1]H	3.117	21.86	9.352	65.60	15.59	109.3	9.050	60.78	27.16	182.3	45.30	303.9
	[1]L	**0.222**	0.844	8.300	24.19	220.1	677.9	**0.957**	3.082	25.19	74.80	660.4	2046
	Ours	0.475	**0.372**	**1.377**	**0.889**	**2.279**	**1.181**	1.339	**0.820**	**3.960**	**1.783**	**6.581**	**2.801**
2^8	[1]H	137.2	983.4	411.5	2950	685.8	4917	398.4	2695	1195	8087	1992	13478
	[1]L	**3.557**	11.19	132.8	411.9	3521	11042	**15.31**	45.34	403.1	1246	$> 10^4$	$> 10^4$
	Ours	7.588	**4.307**	**22.03**	**9.882**	**36.91**	**16.25**	21.42	**8.825**	**63.35**	**23.18**	**106.6**	**38.97**
2^{12}	[1]L	**56.91**	180.4	2124	6657	$> 10^4$	$> 10^5$	**244.9**	742.6	> 6000	$> 10^4$	$> 10^5$	$> 10^5$
	Ours	122.8	**64.42**	**356.7**	**164.7**	**590.6**	**264.8**	346.8	**142.3**	**1026**	**402.7**	**1706**	**657.2**
2^{16}	[1]L	**910.5**	2992	$> 10^4$	$> 10^5$	$> 10^5$	$> 10^6$	**3919**	12017	$> 10^4$	$> 10^5$	$> 10^6$	$> 10^6$
	Ours	1964	**1070**	**5707**	**2765**	**9449**	**4443**	5549	**2366**	**16419**	**6539**	**27289**	**10953**

FPSI for L_∞ Distance. Table 5 shows that our protocol for L_∞ distance achieves a $30 - 305\times$ speedup and reduces communication cost by a factor of $6 - 67\times$ when $d \geq 6$.

Table 5. Communication cost (MB) and running time (s) of FPSI for L_∞ distance.

$m = n$	Protocol	(d, δ)											
		(2,10)		(6,10)		(10,10)		(2,30)		(6,30)		(10,30)	
		Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time
2^4	[1]H	2.073	5.333	18.65	45.70	52.03	122.9	17.55	43.37	158.0	298.1	439.4	759.3
	[1]L	**0.173**	0.660	8.251	24.09	220.0	677.1	**0.517**	1.891	24.75	73.61	660.0	2042
	Ours	0.470	**0.347**	**1.384**	**0.825**	**2.298**	**1.282**	1.340	**0.696**	**3.994**	**1.727**	**6.648**	**2.501**
2^8	[1]H	33.22	78.67	298.8	734.6	833.7	2011	281.0	697.5	2528	4933	> 7000	$> 10^4$
	[1]L	**2.766**	9.047	132.0	408.9	3520	11027	**8.266**	25.10	396.0	1225	$> 10^4$	$> 10^4$
	Ours	7.518	**3.732**	**22.14**	**9.029**	**36.75**	**14.96**	21.44	**7.930**	**63.90**	**22.28**	**106.4**	**36.99**
2^{12}	[1]L	**44.25**	143.4	2112	6612	$> 10^4$	$> 10^5$	**132.3**	420.8	> 6000	$> 10^4$	$> 10^5$	$> 10^5$
	Ours	120.2	**53.74**	**354.1**	**151.1**	**588.0**	**253.2**	343.0	**128.9**	**1022**	**391.4**	**1702**	**644.1**
2^{16}	[1]L	**708.0**	2401	$> 10^4$	$> 10^5$	$> 10^5$	$> 10^6$	**2116**	6796	$> 10^4$	$> 10^5$	$> 10^6$	$> 10^6$
	Ours	1924	**945.6**	**5665**	**2623**	**9408**	**4332**	5488	**2218**	**16358**	**6366**	**27228**	**10779**

9 Conclusion

In this work, we abstract a new primitive called Fmap, which is a powerful technique for FPSI. Many existing FPSI protocols are based on Fmap and their complexity bottlenecks mainly due to high expansion rate of their Fmap instances. We give new constructions of Fmap with small expansion rate for Hamming and $L_{\mathsf{p}\in[1,\infty]}$ distances to break bottlenecks.

We report a generic construction of FPSI from Fmap, which leads to the first FPSI for Hamming distance of which costs are strictly linear with m and n. Meanwhile, we show a construction of mqFRPMT from sUFmap, an enhanced Fmap. We propose an FPSI(-variants) framework from mqFRPMT. Using this framework, we finally get FPSI for $L_{\mathsf{p}\in[1,\infty]}$ distance of which costs scale linearly with anyone of m, n, d, and δ for the first time.

Regarding future works, we present the following thoughts:

- The distinctiveness property of Fmap is intended to make subsequent OKVS encoding possible, which seems unnatural. How to avoid the distinctiveness property to gain a more general abstraction is an interesting question.
- Our FPSI for $L_{\mathsf{p}\in[1,\infty]}$ distance uses $\mathsf{R} \wedge \mathsf{S}$. disj. proj., a stronger assumption than R. disj. proj. of [1], to obtain optimal complexity. Is it possible to construct a protocol under a more realistic assumption (i.e. something weaker than R. disj. proj.) to achieve a near-linear asymptotic complexity and a practical efficiency comparable to the protocol in this work? Any relevant progress would be quite valuable.
- All these protocols above are in the semi-honest setting. We leave the construction of efficient FPSI protocols in the malicious setting as future work.

Acknowledgement. We thank all the anonymous reviewers for helpful feedback. This work is supported by the National Key Research and Development Program of China (2022YFB2701600).

References

1. van Baarsen, A., Pu, S.: Fuzzy private set intersection with large hyperballs. In: Joye, M., Leander, G. (eds.) Advances in Cryptology – EUROCRYPT 2024. pp. 340–369. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-58740-5_12
2. Bienstock, A., Patel, S., Seo, J.Y., Yeo, K.: Near-Optimal oblivious Key-Value stores for efficient PSI, PSU and Volume-Hiding Multi-Maps. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 301–318. USENIX Association, Anaheim, CA (2023)
3. Bienstock, A., Patel, S., Seo, J.Y., Yeo, K.: Batch pir and labeled psi with oblivious ciphertext compression. Cryptology ePrint Archive, Paper 2024/215 (2024), https://eprint.iacr.org/2024/215
4. Chakraborti, A., Fanti, G., Reiter, M.K.: Distance-Aware private set intersection. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 319–336. USENIX Association, Anaheim, CA (2023)
5. Chen, H., Huang, Z., Laine, K., Rindal, P.: Labeled psi from fully homomorphic encryption with malicious security. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1223–1237. CCS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3243836
6. Chen, Y., Zhang, M., Zhang, C., Dong, M., Liu, W.: Private set operations from multi-query reverse private membership test. In: Tang, Q., Teague, V. (eds.) Public-Key Cryptography – PKC 2024. pp. 387–416. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-57725-3_13
7. Chmielewski, L., Hoepman, J.H.: Fuzzy private matching (extended abstract). In: 2008 Third International Conference on Availability, Reliability and Security. pp. 327–334 (2008). https://doi.org/10.1109/ARES.2008.170
8. Chongchitmate, W., Lu, S., Ostrovsky, R.: Approximate psi with near-linear communication. Cryptology ePrint Archive, Paper 2024/682 (2024), https://eprint.iacr.org/2024/682
9. Cong, K., Moreno, R.C., da Gama, M.B., Dai, W., Iliashenko, I., Laine, K., Rosenberg, M.: Labeled PSI from homomorphic encryption with reduced computation and communication. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 1135–1150. CCS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460120.3484760
10. Duong, T., Phan, D.H., Trieu, N.: Catalic: Delegated psi cardinality with applications to contact tracing. In: Moriai, S., Wang, H. (eds.) Advances in Cryptology – ASIACRYPT 2020. pp. 870–899. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-64840-4_29
11. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Cachin, C., Camenisch, J.L. (eds.) Advances in Cryptology - EUROCRYPT 2004. pp. 1–19. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_1
12. Garimella, G., Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: Oblivious key-value stores and amplification for private set intersection. In: Advances in Cryptology – CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II. pp. 395–425. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-84245-1_14

13. Garimella, G., Rosulek, M., Singh, J.: Structure-aware private set intersection, with applications to fuzzy matching. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology – CRYPTO 2022. pp. 323–352. Springer Nature Switzerland, Cham (2022). https://doi.org/10.1007/978-3-031-15802-5_12
14. Garimella, G., Rosulek, M., Singh, J.: Malicious secure, structure-aware private set intersection. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology – CRYPTO 2023. pp. 577–610. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-38557-5_19
15. Indyk, P., Woodruff, D.: Polylogarithmic private approximations and efficient matching. In: Halevi, S., Rabin, T. (eds.) Theory of Cryptography. pp. 245–264. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11681878_13
16. Ion, M., Kreuter, B., Nergiz, A.E., Patel, S., Saxena, S., Seth, K., Raykova, M., Shanahan, D., Yung, M.: On deploying secure computing: Private intersection-sum-with-cardinality. In: 2020 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 370–389 (2020). https://doi.org/10.1109/EuroSP48549.2020.00031
17. Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious prf with applications to private set intersection. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 818–829. CCS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978381
18. Manku, G.S., Jain, A., Das Sarma, A.: Detecting near-duplicates for web crawling. In: Proceedings of the 16th International Conference on World Wide Web. p. 141-150. WWW '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1242572.1242592
19. Mohammadi-Kambs, M., Hölz, K., Somoza, M.M., Ott, A.: Hamming distance as a concept in dna molecular recognition. ACS Omega **2**(4), 1302–1308 (2017). https://doi.org/10.1021/acsomega.7b00053
20. Patra, A., Schneider, T., Suresh, A., Yalame, H.: ABY2.0: Improved Mixed-Protocol secure Two-Party computation. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2165–2182. USENIX Association (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/patra
21. Raghuraman, S., Rindal, P.: Blazing fast psi from improved okvs and subfield vole. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2505–2517. CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3548606.3560658
22. Rindal, P., Schoppmann, P.: Vole-psi: Fast oprf and circuit-psi from vector-ole. In: Canteaut, A., Standaert, F.X. (eds.) Advances in Cryptology – EUROCRYPT 2021. pp. 901–930. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-77886-6_31
23. Uzun, E., Chung, S.P., Kolesnikov, V., Boldyreva, A., Lee, W.: Fuzzy labeled private set intersection with applications to private Real-Time biometric search. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 911–928. USENIX Association (2021)
24. Uzun, E., Yagemann, C., Chung, S., Kolesnikov, V., Lee, W.: Cryptographic key derivation from biometric inferences for remote authentication. In: Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security. p. 629-643. ASIA CCS '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3433210.3437512

25. Wu, M., Yuen, T.H.: Efficient unbalanced private set intersection cardinality and user-friendly privacy-preserving contact tracing. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 283–300. USENIX Association, Anaheim, CA (2023)
26. Ye, Q., Steinfeld, R., Pieprzyk, J., Wang, H.: Efficient fuzzy matching and intersection on private datasets. In: Lee, D., Hong, S. (eds.) Information, Security and Cryptology – ICISC 2009. pp. 211–228. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14423-3_15

FOLEAGE: $\mathbb{F}_4$OLE-Based Multi-party Computation for Boolean Circuits

Maxime Bombar[1,2(✉)], Dung Bui[3], Geoffroy Couteau[4], Alain Couvreur[5], Clément Ducros[6], and Sacha Servan-Schreiber[7]

[1] Cryptology Group, CWI, Amsterdam, The Netherlands
maxime.bombar@cwi.nl
[2] Institut de Mathématiques de Bordeaux, Bordeaux, France
[3] IRIF, Université Paris Cité, Paris, France
bui@irif.fr
[4] CNRS, IRIF, Université Paris Cité, Paris, France
couteau@irif.fr
[5] INRIA, Laboratoire LIX, École Polytechnique, Institut Polytechnique de Paris, Palaiseau, France
alain.couvreur@inria.fr
[6] IRIF, Université Paris Cité, INRIA, Paris, France
cducros@irif.fr
[7] MIT, Cambridge, MA, USA
3s@mit.edu

Abstract. Secure Multi-party Computation (MPC) allows two or more parties to compute any public function over their privately-held inputs, without revealing any information beyond the result of the computation. Modern protocols for MPC generate a large amount of input-independent preprocessing material called *multiplication triples*, in an offline phase. This preprocessing can later be used by the parties to efficiently instantiate an input-dependent online phase computing the function.

To date, the state-of-the-art secure multi-party computation protocols in the preprocessing model are tailored to secure computation of *arithmetic* circuits over large fields and require little communication in the preprocessing phase, typically $O(N \cdot m)$ to generate m triples among N parties. In contrast, when it comes to computing preprocessing for computations that are naturally represented as *Boolean* circuits, the state-of-the-art techniques have not evolved since the 1980s, and in particular, require every pair of parties to execute a large number of oblivious transfers before interacting to convert them to N-party triples, which induces an $\Omega(N^2 \cdot m)$ communication overhead.

In this paper, we introduce $\mathbb{F}_4$OLEAGE, which addresses this gap by introducing an efficient preprocessing protocol tailored to Boolean circuits, with semi-honest security and tolerating $N-1$ corruptions. $\mathbb{F}_4$OLEAGE has excellent concrete performance: It generates m multiplication triples over $\mathbb{F}_2$ using only $N \cdot m + O(N^2 \cdot \log m)$ bits of communication for N-parties, and can concretely produce over 12 million triples per second in the 2-party setting on one core of a commodity machine. Our result builds upon an efficient Pseudorandom Correlation Generator (PCG) for multiplica-

K.-M. Chung and Y. Sasaki (Eds.): ASIACRYPT 2024, LNCS 15489, pp. 69–101, 2025.
https://doi.org/10.1007/978-981-96-0938-3_3

tion triples over the field $\mathbb{F}_4$. Roughly speaking, a PCG enables parties to stretch a short seed into a large number of pseudorandom correlations *non-interactively*, which greatly improves the efficiency of the offline phase in MPC protocols. This is achieved by introducing a number of protocol-level, algorithmic-level, and implementation-level optimizations on the recent PCG construction of Bombar et al. (Crypto 2023) from the Quasi-Abelian Syndrome Decoding assumption.

1 Introduction

A secure multiparty computation (MPC) protocol for a public functionality f allows N parties with private inputs $(x_1, \cdots, x_N)$ to securely compute $f(x_1, \cdots, x_N)$, while concealing all other information about their private inputs to coalitions of corrupted parties. MPC was introduced in the seminal work of Goldreich, Micali, and Wigderson [29] (GMW), and has since led to a rich body of work developing the foundations of MPC, and even practical open-source libraries [34].

Two of the leading paradigms in secure computation are garbled circuits [45] and secret-sharing-based secure computation [29]. The seminal GMW protocol is of the latter type. In a secret-sharing-based MPC protocol, the parties hold shares of the inputs and iteratively compute the circuit representing the function, gate-by-gate. Because addition gates can be computed locally by the parties holding the input shares, only multiplication gates require interaction between the parties to evaluate. As such, the major bottleneck of MPC protocols is due to the communication required to evaluate the multiplication gates in a circuit. (Note that this is also true of the garbled circuit approach where addition gates are "free" and only multiplication gates need to be garbled [37]).

However, a core advantage of secret-sharing-based MPC, first identified in the work of Beaver [5], is that secure multiplications can be *preprocessed* in an *input-independent* precomputation phase. In particular, the parties can securely generate additive shares of many "Beaver triples" $(a, b, a \cdot b) \in \mathbb{F}^3$. Then, for each multiplication gate that needs to be computed in the online phase, the parties can run a fast information-theoretically secure multiplication protocol that consumes one Beaver triple and involves communicating just two elements of $\mathbb{F}$ per party. This model of secure computation with preprocessing forms the basis for modern MPC protocols due to the efficiency of the online phase. However, this preprocessing paradigm only serves to push the inefficiency bottleneck of MPC to the offline phase that consists of generating many Beaver triples. We briefly survey the different techniques that have been developed in the last couple of decades for the efficient generation of Beaver triples in an MPC setting.

Modern Secure Computation Protocols. The traditional approach for securely generating Beaver triples relies on Oblivious Transfers (OT) [25,40]: an N-party Beaver triple over $\mathbb{F}$ is generated by letting each *pair* of parties execute $\log |\mathbb{F}|$ oblivious transfers [26], and thanks to OT *extension* protocols [6,33], generating a large number of OTs requires only cheap symmetric-key operations. This OT-based approach is very competitive with a small number of parties, but becomes very inefficient with many parties. Specifically, because *each pair* of parties needs to perform OTs, the communication and computation costs are on the order of $\Omega(N^2)$, which quickly becomes impractical as N grows large.

Over the past decade, the practicality of secure computation has increased tremendously [22,23,31,34–36]. This is especially true in the setting of secure computation of *arithmetic circuits over large fields*. Starting with the celebrated SPDZ protocol [23], a sequence of works has developed fast protocols that use Ring-LWE-based somewhat homomorphic encryption, or even linearly homomorphic encryption, to generate m Beaver triples with only $O(m \cdot N)$ communication and computation per triple. These approaches significantly improve over the "naïve" $\Omega(m \cdot N^2)$ cost of the OT-based approach. Over sufficiently large fields (e.g., larger than 2^λ), when generating many triples, state-of-the-art protocols such as Overdrive [36] achieve very good concrete efficiency.

More recently, following the line of work on silent secure computation initiated in [9,11,12], Boyle et al. [14] have shown how to generate a large number m of *pseudorandom* (as opposed to truly random) Beaver triples under the Ring-LPN assumption. Their approach uses $O(\log m \cdot N^2)$ communication, followed solely by *local* computation, with good concrete efficiency (the authors estimated a throughput of around 10^5 triples per second on one core of a standard laptop). For sufficiently large values of m, this is highly competitive with Overdrive. However, both Overdrive and the existing PCG-based approach share a common restriction: they are only usable over large fields.

Secure Computation of Boolean Circuits. In contrast to the secure computation of arithmetic circuits over large fields, the fastest way to run N-party MPC protocols for Boolean circuits remains the "naïve" method of generating many pairwise OTs, at a cost of $\Omega(m \cdot N^2)$ bits for m Beaver triples. This is in contrast to the *two-party* setting, where two-party Beaver triples can be generated very efficiently thanks to a recent line of work [9,11,12] on silent OT extension. In silent OT extension, two parties can generate m Beaver triples using only $O(\log m)$ communication. The state-of-the-art protocols in this area [10,21,41] achieve impressive throughputs of several million Beaver triples per second on one core of a standard laptop. Furthermore, the recent SoftSpoken OT extension protocol [43] yields even faster OTs at the cost of increasing communication. For example, SoftSpoken can generate nearly 30M OT/s on `localhost` at the cost of increasing the communication to $64m$ bits to generate m Beaver triples; other communication/computation tradeoffs are possible [43, Table 1].[1]

The situation, however, is much less satisfying for the setting of secure computation of Boolean circuits with a larger number of parties. Protocols such as SPDZ [23] and Overdrive [36] do not perform well when generating Beaver triples for Boolean circuits, even in the passive setting. This is due to the high overhead of embedding $\mathbb{F}_2$ in an extension field compatible with the number theoretic-transform used in efficient instantiations of the BGV encryption scheme [17]. Furthermore, silent OT extension techniques build on Pseudorandom Correlation Generators (PCGs), which typically work only in the two-party setting [12].

[1] Note that we need two calls to the OT functionality to generate one Beaver triple.

To handle more parties, one needs the stronger notion of *programmable* PCG [14], which, informally, allows partially specifying parts of the generated correlation. Unfortunately, while efficient programmable PCGs over large fields were introduced in [14], building *concretely efficient*, programmable PCGs over $\mathbb{F}_2$ has remained elusive thus far, making N-party PCGs for $\mathbb{F}_2$ primarily of theoretical interest. The state-of-the-art is the recent work of Bombar et al. [8], which generates Beaver triples over any field $\mathbb{F}_q$ with $q \geq 3$. However, Bombar et al. [8] leave analyzing the concrete efficiency for future work.

In light of this state of affairs, to the best of our knowledge, the current most efficient approach for N-party secure computation of Boolean circuits remains the classical OT-based approach. In a little more detail, to generate each Beaver triple, each party P_i samples a random pair (a_i, b_i) of bits, and each pair (P_i, P_j) of parties executes two oblivious transfer protocols to generate additive shares of $a_i b_j$ and $a_j b_i$. Then, all parties aggregate their shares to obtain shares of $\sum_{i,j} a_i b_j = (\sum_i a_i) \cdot (\sum_j b_j)$. When generating m Beaver triples, this approach requires $N \cdot (N-1) \cdot m$ oblivious transfers in total (to be compared with the $O(N^2 \cdot \log m)$ communication of [14], or the $O(N \cdot m)$ communication of Overdrive [36], for the case of arithmetic circuits over large fields). While there has been tremendous progress in constructing efficient OT protocols [33,43], even using silent OT extension (which has the lower communication overhead) requires $3N \cdot (N-1) \cdot m$ bits of communication (ignoring some $o(m)$ terms). Using SoftSpoken OT [43] instead, which appears to be the most computationally efficient solution, and setting the "communication/computation tradeoff" parameter k to $k = 5$, the communication increases to $32N \cdot (N-1) \cdot m$ bits. When the number of parties grows, this soon becomes very inefficient.[2]

1.1 Our Focus and Contributions

In this paper, we focus on secure computation of general Boolean circuits with multiple parties in the semi-honest setting. Our main contribution is $\mathbb{F}_4$OLEAGE, a novel $\mathbb{F}_4$-OLE-based protocol for secure computation *in the preprocessing model* that significantly outperforms the state-of-the-art approach in both the *two-party* and *multi-party* setting. In particular, $\mathbb{F}_4$OLEAGE enjoys much lower communication in the preprocessing phase than all known alternatives and has a very low computational overhead. We expect $\mathbb{F}_4$OLEAGE to be the fastest alternative for large enough circuits on almost any realistic network setting, for any number of parties between two and several hundred. $\mathbb{F}_4$OLEAGE builds upon recent results constructing efficient PCGs and introduces several protocol-level, algorithmic-level, and implementation-level optimizations to make these PCG constructions blazing fast (see Sect. 5 for a performance evaluation).

In the Two-Party Setting. ($N = 2$), $\mathbb{F}_4$OLEAGE enjoys a silent preprocessing (generating m multiplication triples requires $O(\log m)$ communication), and

[2] For a very large number of parties, the linear scaling in N of Overdrive should become favorable. However, after private communication with the authors of Overdrive, the break-even point for communication seems to happen only for values of N in the range of 400+, due to the high overhead of using BGV and embedding $\mathbb{F}_2$ elements.

significantly outperforms all previous silent protocols. In particular, our implementation generates around 12.3 million Beaver triples *per second* on one core of an Amazon `c5.metal` server. Compare this to the state-of-the-art silent OT protocol RRT [41] which generates 3.4 million Beaver triples per second with the same setup. This makes RRT more than 3.5 times slower compared to $\mathbb{F}_4$OLEAGE. The fastest *non-silent* OT protocol, SoftSpoken OT, generates around 26 million multiplication triples per second on `localhost` in its fastest regime (using $k = 2$ [43, Table 1]), while requiring around $128 \cdot m$ bits of total communication. However, while our approach does achieve a blazing-fast throughput, it has some limitations. In particular, the preprocessing phase of $\mathbb{F}_4$OLEAGE requires more rounds (16 rounds instead of 3 for generating 26M triples compared to [43]). Additionally, our seed size is roughly $130\times$ larger compared to [41], and $2\times$ larger compared to [14]. This makes $\mathbb{F}_4$OLEAGE less suitable for generating a small number of triples. Eventually, our protocols are tailored to the generation of multiplication triples over $\mathbb{F}_2$ in the semi-honest setting: their efficiency scales less favorably in other settings, such as generating string OTs or authenticated triples.

In the Multi-party Setting. $(N > 2)$, $\mathbb{F}_4$OLEAGE achieves *almost-silent* preprocessing: to securely compute a circuit with m AND gates, following a silent phase with $O(N^2 \cdot \log m)$ communication, our preprocessing phase requires a single broadcast of $N \cdot m$ bits (one bit per AND gate and per party), and the online phase is the standard GMW protocol. As N grows, this represents a drastic reduction in communication compared to the $\sim 3 \cdot N^2 m$ communication obtained when using silent OT extension, or the $\sim 32 \cdot N^2 m$ communication obtained with SoftSpoken OT, while remaining highly competitive in terms of computation.

Comparison with the State of the Art. In Table 1, we provide a comparison between $\mathbb{F}_4$OLEAGE, SoftSpoken, and RRT, for $N = 10$ and $N = 2$ parties. In the multiparty setting, due to the very low bandwidth requirement of $\mathbb{F}_4$OLEAGE, we observe that computation is systematically the bottleneck when evaluated on one core of a commodity server. This indicates that $\mathbb{F}_4$OLEAGE is likely to stand out even more whenever more computational power is available, e.g., when evaluated in parallel on multiple cores.

The numbers in Table 1 have been computed using the running time T measured for generating 3^{16} OLEs (Table 4, using the noise parameter $t = 27$ and $c = 3$) on one core of AWS `c5.metal`, and estimating the per-party cost to generate 10^9 N-party Beaver triples as $2 \cdot (N-1) \cdot T \cdot (10^9/3^{16})$. When $N = 2$, the cost is estimated as $T \cdot (10^9/3^{16})$, accounting for the factor-2 saving tailored to the 2-party setting. For communication, we computed an estimate of $C =$ 13MB of communication for our distributed protocol for generating a seed for 3^{18} OLEs. While one could in principle directly generate a seed that stretches to 10^9 OLEs, this would significantly slow down the computation as the 10^9 OLEs must be expanded all at once, and would not fit in memory. Hence, we estimate the communication as $2 \cdot (N-1) \cdot (3^{18}/10^9) \cdot C$ for generating 10^9 N-party Beaver

Table 1. Comparison of state-of-the-art protocols to generate N-party Beaver triples over $\mathbb{F}_2$ for $N = 10$ and $N = 2$ parties. The `localhost` column reports the runtimes (ignoring communication) for generating 10^9 triples. All protocols run on one core of AWS `c5.metal` (3.4GHz CPU); all runtimes averaged across ten trials. "Communication" denotes the number of bits communicated per party for 10^9 triples. LAN and WAN refer to the theoretical time required to generate 10^9 triples over a 1 Gbps and 100 Mbps network respectively, with respective delays 1ms and 40ms. Numbers in **bold red** indicate that the bottleneck cost is the local computation. *Maximum theoretical throughput with more computational power (e.g., using multiple cores). Since each party computes $2 \cdot (N-1)$ expansions for the PCG in parallel for an N-party Beaver triple, the running time is divided by C when using C cores whenever $C \leq 2 \cdot (N-1)$.

	Communication	`localhost`	**LAN**	**WAN**
Multi-party setting ($N = 10$)				
SoftSpoken ($k = 2$)	134 GB	342s	1192s	12207s
SoftSpoken ($k = 4$)	67 GB	405s	596s	6104s
SoftSpoken ($k = 8$)	34 GB	1900s	**1900s**	3052s
			*298s	
RRT	6.3 GB	2619s	**2619s**	**2619s**
			*50.3s	*515s
$\mathbb{F}_4$OLEAGE	0.7 GB	1463s	**1463s**	**1463s**
			*5.6s	*57.9s
Two-party setting ($N = 2$)				
SoftSpoken ($k = 2$)	15 GB	38s	119s	1221s
SoftSpoken ($k = 4$)	7.5 GB	45s	60s	610s
SoftSpoken ($k = 8$)	3.7 GB	211s	**211s**	**211s**
RRT	258 KB	292s	**292s**	**292s**
$\mathbb{F}_4$OLEAGE	33.5 MB	81s	**81s**	**81s**

triples (as 3^{18} OLEs is the maximum expansion size we could fit in the memory), and an additional 10^9 bits of communication per party (in the setting $N > 2$).

Security. The security of $\mathbb{F}_4$OLEAGE relies on the Quasi-Abelian Syndrome Decoding (QA-SD) assumption, a variant of the syndrome decoding assumption that was recently introduced in [8]. QA-SD is a generalization of the standard quasi-cyclic syndrome decoding assumption (used in many previous works [1–3,11]) which was shown to asymptotically resist all known attacks against LPN and syndrome decoding in [8]. As a contribution of independent interest, we complement their preliminary analysis with thorough concrete cryptanalysis of the security of QA-SD against *all* state-of-the-art attacks. Our analysis covers in full detail the distribution of the noisy coordinates under folding attacks and the cost of attacking folded QA-SD instances using tailored Information Set Decoding (ISD) algorithms over $\mathbb{F}_4$. We include the SageMath script used to select our parameters from this analysis. As a byproduct, our precise analysis yields an attack undermining the claimed security of the parameters from [8]. Specifically, with our attack and a set of parameters $c = 4, t = 16$ (see the full version for details), [8] can only achieve a security level of 118 bits instead of 128 bits. This could probably be improved.

Implementation. We provide an open-source prototype implementation of our PCG construction in C. Our implementation is covered in detail in Sect. 5. It includes, in particular, a new implementation of distributed point functions that work with a ternary input domain (providing faster evaluation at a slightly increased key size), and optimized FFT over $\mathbb{F}_4$. We cover all these contributions in more detail in Sect. 2.

An Alternative Approach. Our construction is not the only way of generating $\mathbb{F}_2$-Beaver triples. We note here that one could make use of the so-called *Reverse Multiplication Friendly Embeddings* (RMFE), as introduced in [18] to turn Beaver triples over a large field $\mathbb{F}_{2^m}$ into Beaver triples over $\mathbb{F}_2$. We briefly remark on why such an approach will be considerably less efficient than ours, while still relying on similar LPN-style assumptions. More precisely, a (k, m)-RMFE is a pair of maps (ϕ, ψ) with $\phi\colon \mathbb{F}_2^k \rightarrow \mathbb{F}_{2^m}$ and $\psi\colon \mathbb{F}_{2^m} \rightarrow \mathbb{F}_2^k$ such that for any $\mathbf{x}, \mathbf{y} \in \mathbb{F}_2^k$, we have $\mathbf{x} \star \mathbf{y} = \psi(\phi(\mathbf{x})\phi(\mathbf{y}))$, where $\star$ denotes the component-wise product. Using parameters from [18, Remark 7], there exist a $(12, 33)$-RMFE. Now, let us assume that the parties want to generate s Beaver triples over $\mathbb{F}_2$ (where $s < 12 \times 2^{33}$ using their parameters). They can proceed as follows:

1. First, the parties use a programmable PCG for OLEs over large fields to generate $s/12$ random Beaver triples over $\mathbb{F}_{2^{33}}$. Note that $s/12 < 2^{33}$.
2. Second, for each triple (a, b, c), the participant locally sample random elements seen as shares of random $\mathbf{u}, \mathbf{v} \in \mathbb{F}_2^{12}$, and locally apply ϕ to get shares of $\phi(\mathbf{u})$ and $\phi(\mathbf{v})$.
3. The parties consume (a, b, c) to reconstruct $a + \phi(\mathbf{u})$ and $b + \psi(\mathbf{v})$, and then locally get additive shares of $\phi(\mathbf{u}) \cdot \phi(\mathbf{v})$.
4. Finally, the parties can locally apply ψ to get shares of $\mathbf{u} \star \mathbf{v}$.

This approach produces $k = 12$ times as many random Beaver triples over $\mathbb{F}_2$ as triples originally generated over $\mathbb{F}_{2^{33}}$. However, Step 3 communicates two elements of $\mathbb{F}_{2^{33}}$. In other words, this approach requires on average $66/12 = 5.5$ times more communication than $\mathbb{F}_4$OLEAGE in the offline phase.

The cost of computation is not clear since there is no implementation of a PCG for OLEs over $\mathbb{F}_{2^{33}}$. But assuming that a suitable multiplication algorithm is designed, and using the numbers from [14], a ballpark estimate would be around $100k$ OLEs per second. In the two party setting, this amounts to $50k \times 12 = 600k$ Beaver triples over $\mathbb{F}_2$, which is likely to be an overestimate. Overall, this makes $\mathbb{F}_4$OLEAGE at least concretely 20 times faster.

Full Version of this Work. More technical details are provided in the full version of the paper [7].

2 Technical Overview

In this section, we provide a detailed description of our results and the main technical ideas underlying them. In Sect. 2.1, we provide background on secure multi-party computation realized from PCGs for OLE correlations. In Sect. 2.2 we describe the PCG construction of [8], which forms the basis for our preprocessing protocol. In Sect. 2.3, we describe our idea for converting $\mathbb{F}_4$ triples into $\mathbb{F}_2$ triples, which we tailor to the two-party case in Sect. 2.4. In Sect. 2.5, we describe our optimized PCG construction. In Sect. 2.6, we explain how we can obtain an efficient distributed seed generation protocol for our PCG construction. Finally, in Sect. 2.7, we overview our improved analysis of the $\mathsf{QA\text{-}SD}$ assumption.

Notations. Unless otherwise stated, an N-party linear secret shares of a value v is denoted $[\![v]\!] = ([\![v]\!]_1, \dots, [\![v]\!]_N)$, where the i-th party obtains share $[\![v]\!]_i$. To disambiguate shares over $\mathbb{F}_4$ and shares over $\mathbb{F}_2$, we denote the field size with a superscript, i.e., $[\![\cdot]\!]^4$ and $[\![\cdot]\!]^2$, respectively. We identify $\mathbb{F}_4$ with $\mathbb{F}_2[X]/(X^2 + X + 1)$ and let θ denote a primitive root of $X^2 + X + 1$. Given an element $x \in \mathbb{F}_4$, we write $x(0)$ and $x(1)$ to denote the $\mathbb{F}_2$-coefficients of x viewed as a polynomial over $\mathbb{F}_2[X]/(X^2 + X + 1)$; that is, $x = x(0) + \theta \cdot x(1)$. Additional notation can be found in Sect. 3.

2.1 Background: Secure MPC from PCGs

We start by describing prior approaches to realizing MPC in the preprocessing model from PCGs for OLE correlations.

PCGs for the OLE Correlation. Our starting point is the template for generating N-party pseudorandom Beaver triples put forth by Boyle et al. [14]. At the heart of their framework is the use of a programmable PCG [14] for the OLE correlation. Concretely, a PCG for a target correlation C (i.e., a distribution over pairs of strings) is a pair of algorithms $(\mathsf{PCG.Gen}, \mathsf{PCG.Eval})$ such that

- $\mathsf{PCG.Gen}$ generates a pair of *succinct* keys $(\mathsf{k}_0, \mathsf{k}_1)$ jointly encoding the target correlation, and
- $\mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma)$ produces a string R_σ corresponding to party σ's secret share of the target correlation.

At a high level, a PCG must satisfy two properties: (1) *pseudorandomness* (or correctness) which states that (R_0, R_1) must be indistinguishable from a random sample from C, and (2) *security* which states that R_σ should appear random conditioned on satisfying the target correlation with $R_{1-\sigma} = \mathsf{PCG.Expand}(1 - \sigma, \mathsf{k}_{1-\sigma})$ even given $\mathsf{k}_{1-\sigma}$, for $\sigma \in \{0, 1\}$.

We focus on the OLE correlation over a finite field $\mathbb{F}$. For a length-m OLE correlation, the string R_0 (which we call the *sender* output) is a list of m tuples $(u_i, v_i)_{i \le m} \in (\mathbb{F}^2)^m$, and the string R_1 (which we call the *receiver* output) is a list of m pairs $(x_i, w_i)_{i \le m} \in (\mathbb{F}^2)^m$ such that $w_i = u_i \cdot x_i + v_i$ for every i. Observe that, we can equivalently view v_i and $-w_i$ as additive shares of $u_i \cdot x_i$, which we will denote as $[\![u_i \cdot x_i]\!]$. Informally, security for the OLE correlation amounts to showing that the following two properties hold:

– **Sender security:** from the viewpoint of the receiver (who has k_1 and generates (x_i, w_i)), the distribution of (u_i, v_i) is computationally indistinguishable from the distribution of $(u_i, w_i - u_i \cdot x_i)$, for a uniformly random $u_i \leftarrow_R \mathbb{F}$.
– **Receiver security:** from the viewpoint of the sender (who has k_0), the distribution of each x_i is computationally indistinguishable from a random field element.

Going from OLE to Beaver Triples. As shown in [12], given a PCG for the OLE correlation (or a PCG for OLE for short), two parties can generate many pseudorandom Beaver triples over $\mathbb{F}$ as follows. First, the parties compute $\mathsf{PCG.Gen}$ via a two-party secure computation protocol to obtain PCG keys k_0 and k_1, respectively. Then, using $\mathsf{PCG.Expand}$, the two parties locally obtain many correlations of the form $(u_i, [\![u_i x_i]\!]_0)$ and $(x_i, [\![u_i x_i]\!]_1)$, respectively. Given two such OLE correlations, where one party has $(u_0, u_1, [\![u_0 x_0]\!]_0, [\![u_1 x_1]\!]_0)$ and the other party has $(x_0, x_1, [\![u_0 x_0]\!]_1, [\![u_1 x_1]\!]_1)$, the two parties can *locally* derive one Beaver triple of the form $([\![a]\!], [\![b]\!], [\![ab]\!])$ by computing:

$$(\underbrace{[\![u_0 + x_1]\!]}_{[\![a]\!]}, \underbrace{[\![u_1 + x_0]\!]}_{[\![b]\!]}, [\![u_0 x_0 + u_1 x_1]\!] + u_0 u_1 + x_0 x_1 = \underbrace{[\![(u_0 + x_1) \cdot (u_1 + x_0)]\!]}_{[\![ab]\!]}).$$

In a little more detail, the sender computes their share of the Beaver triple as $(u_0, u_1, [\![u_0 x_0]\!]_0 + [\![u_1 x_1]\!]_0 + u_0 u_1)$ and the receiver computes their share as $(x_1, x_0, [\![u_0 x_0]\!]_1 + [\![u_1 x_1]\!]_1 + x_0 x_1)$. While this technique works well in the two-party setting, in the *multi*-party setting, things are not so simple.

Going from Two Parties to Many Parties. As first discussed by Boyle et al. [14], to generate N-party Beaver triples using a PCG for OLE, the parties need to ensure *consistency* among the OLE correlations generated by *each pair of parties*. That is, to generate one multiplication triple $([\![a]\!], [\![b]\!], [\![ab]\!])$, we need each pair of parties (P_i, P_j) to hold respective values (a_i, b_i) and (a_j, b_j) (viewed as an individual share of a and b), together with two-party shares $[\![a_i b_j]\!]$ and $[\![a_j b_i]\!]$. Then, all parties can combine their shares to get

$$[\![(\textstyle\sum_i a_i) \cdot (\sum_j b_j\ )]\!] = \textstyle\sum_{i \neq j} [\![a_i b_j]\!] + \sum_i a_i b_i.$$

Observe that this requires party P_i to have OLEs of the form $(a_i, [\![a_i a_j]\!]_i)$, with every other party P_j (who in turn has share $(a_j, [\![a_i a_j]\!]_j)$), *where P_j's value a_i remains the same across all OLEs.* This is precisely what the notion of a *programmable* PCG for OLE achieves: it allows the parties to specify seeds (ρ_0, ρ_1) such that $\mathsf{PCG.Gen}(\rho_0, \rho_1)$ outputs keys $\mathsf{k}_0, \mathsf{k}_1$ that, informally speaking, have all the pseudorandom (a_i, b_i) deterministically generated from the seeds ρ_0 and ρ_1 respectively (while still maintaining the required security properties). By reusing the same seeds across executions with multiple parties, the parties can ensure the required consistency across their outputs.

2.2 Constructing Programmable PCGs

In addition to defining the notion of programmable PCGs, the work of Boyle et al. [12,14] introduced a construction from a variant of the LPN assumption over rings. At a high level, the ring-LPN assumption they introduce states that $(a, as + e)$ is hard to distinguish from (a, b), where a, b are random polynomials from a suitable ring $\mathcal{R} = \mathbb{F}_q[X]/(P(X))$, where P splits into $\deg(P)$ linear factors and s, e are random *sparse* polynomials from $\mathcal{R}$. The construction of Boyle et al. proceeds by generating a single large pseudorandom OLE correlation over a polynomial ring $\mathcal{R} = \mathbb{F}_q[X]/(P(X))$, assuming the hardness of the ring-LPN assumption over $\mathcal{R}$. When P splits into $D = \deg(P)$ linear factors, the Chinese Remainder Theorem makes it possible to convert this large OLE correlation over $\mathcal{R}$ into D OLE correlations over $\mathbb{F}_q$ (by reducing it modulo each of the factors of P). Unfortunately, the condition that P splits requires $|\mathbb{F}_q| \geq D$, which restricts the construction to only work over large fields. This makes the resulting OLE correlations only suitable for generating Beaver triples over $\mathbb{F}_q$, which limits their applications. Moreover, other existing efficient (non-PCG-based) protocols for generating Beaver triples are also restricted to large fields [23,36]. However, for the Boolean circuit case, the state-of-the-art remains the basic OT-based approach originally proposed in the GMW protocol.

A Programmable PCG for $\mathbb{F}_4$-OLE. The large-field restriction of the Boyle et al.'s PCG construction was recently overcome by Bombar et al. [8]. At a high-level, the authors of [8] manage to replace the polynomial ring $\mathcal{R}$ by a suitable *Abelian group algebra* $\mathbb{F}[\mathbb{G}]$ (that is, the set of formal sums $\sum_{g\in\mathbb{G}} a_g g$ for $a_g \in \mathbb{F}$, where $\mathbb{G}$ is an Abelian group; endowed with the convolution product), which identifies to some ring of multivariate polynomials. Moreover, they show that an appropriate choice of Abelian group algebra can simultaneously satisfy the following properties, for almost every choice of finite field $\mathbb{F}$:

1. $\mathbb{F}[\mathbb{G}]$ is isomorphic to many copies of $\mathbb{F}$ (note that this property is necessary to convert an OLE correlation over $\mathbb{F}[\mathbb{G}]$ into many OLEs over $\mathbb{F}$),
2. The assumption that $(a, as + e)$ is indistinguishable from random over $\mathbb{F}[\mathbb{G}] \times \mathbb{F}[\mathbb{G}]$, with $a \stackrel{\$}{\leftarrow} \mathbb{F}[\mathbb{G}]$ and (s, e) two random *sparse* elements of $\mathbb{F}[\mathbb{G}]$ (with respect to the canonical notion of sparsity over the group algebra, i.e., sparse formal sums $\sum_{g\in\mathbb{G}} a_g g$) is a plausible assumption,
3. Operations over $\mathbb{F}[\mathbb{G}]$ can be computed efficiently using a Fast Fourier Transform (FFT) algorithm [8,38].

The second property is a new variant of the syndrome decoding (or LPN) assumption which the authors called *Quasi-Abelian Syndrome Decoding*. It naturally extends to a "module"-variant, i.e., the indistinguishability of pairs $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s}\rangle + e)$ where $\mathbf{s}$ and e are drawn from a sparse distribution, and generalizes both the quasi-cyclic syndrome decoding (when $\mathbb{G}$ is a cyclic group), and the LPN or syndrome decoding assumption (when $\mathbb{G} = \{1\}$). The work of Bombar et al. [8] also provides extensive support for this assumption by showing that it resists all *linear attacks*, a class of attacks capturing the most known attacks on the LPN assumption and its variants, and proposes a set of parameters resisting all concrete attacks known at that time. The combination of these three properties allowed them to build an efficient programmable PCG for OLEs over $\mathbb{F}$.

Despite the progress made in [8], their programmable PCG construction is limited in that it applies only to generating OLE correlations over all finite fields $\mathbb{F}$ *except for* $\mathbb{F}_2$. This stems from the fact that there does not exist any group $\mathbb{G}$ such that $\mathbb{F}_2[\mathbb{G}]$ is isomorphic to $\mathbb{F}_2^n$ for $n > 1$ (see [8, Theorem 47]). In contrast, the case of $\mathbb{F}_2$, is *precisely* the case that we are interested in when considering Boolean circuits, which require generating Beaver triples over $\mathbb{F}_2$.

Additionally, the *concrete* efficiency of an FFT computed over the group algebra remains unclear, since Bombar et al. left estimating the performance of FFTs on $\mathbb{F}[\mathbb{G}]$ for future work. As such, the concrete efficiency of their programmable PCG construction is unknown, making it difficult to determine whether or not it is sufficiently efficient to be applied in practical applications (all other components of their construction consist of standard tools used in the PCG literature, which are known to have concretely efficient implementations).

Our Contribution. Looking ahead, our main contribution is to build upon the work of Bombar et al. through a number of simple yet powerful observations that allow us to arrive at an efficient PCG for Beaver triples, suitable for use in secure multi-party computation of Boolean circuits.

- First, we show that we can use their programmable PCG for generating OLEs *over* $\mathbb{F}_4$ to generate multiplication triples *over* $\mathbb{F}_2$, sidestepping the "$\mathbb{F}_2$ barrier" of their PCG construction, at the cost of a *single bit of communication* per triple and per party in the preprocessing phase, or even without any communication when $N = 2$.
- Second, we introduce a number of concrete optimizations to the PCG construction of Bombar et al. [8] that are tailored to the special case of $\mathbb{F} = \mathbb{F}_4$, which gives us an incredibly efficient programmable PCG over $\mathbb{F}_4$. Compared with the fastest previous programmable PCGs of [14], our optimized implementation shows that our construction is *two orders of magnitude* faster.
- Third, we give a much more in-depth cryptanalysis of the QA-SD assumption, closely analyzing all known attacks in the literature, and showing that the set of parameters proposed in [8] should be reduced by at least 10 bits. To facilitate future cryptanalysis of the QA-SD assumption, in the full version [7] we carefully overview all known attacks and assumptions, and provide a script for automatically calculating parameters.

In the next few subsections, we provide more details on the above contributions.

2.3 $\mathbb{F}_2$-Triples from $\mathbb{F}_4$-Triples

Since $\mathbb{F}_4$ is an extension field of $\mathbb{F}_2$, a Boolean circuit can be viewed as an $\mathbb{F}_4$-arithmetic circuit. Hence, using an OLE correlation over $\mathbb{F}_4$ to construct N-party Beaver triples over $\mathbb{F}_4$ directly yields an MPC protocol for *Boolean* circuits in the preprocessing model via the GMW template [29]. Unfortunately, compared to using $\mathbb{F}_2$-Beaver triples, the communication in the *online* phase is doubled, because each party has to send two elements of $\mathbb{F}_4$ per AND gate, hence 4 bits instead of 2 with GMW.

Our core observation is that one can make much better use of these N-party multiplication triples over $\mathbb{F}_4$: we show how to convert an $\mathbb{F}_4$-multiplication triple into an $\mathbb{F}_2$-multiplication triple using *a single bit of communication* per party. Once converted into $\mathbb{F}_2$-triples, these triples can be used within the standard GMW protocol that communicates two bits per party and per AND gate in the online phase. To explain the observation, let $([\![a]\!]^4, [\![b]\!]^4, [\![ab]\!]^4)$ be a Beaver triple over $\mathbb{F}_4$. Writing $x = x(0) + \theta \cdot x(1)$ for any $x \in \mathbb{F}_4$, with θ a root of the polynomial $X^2 + X + 1$ (hence $\theta^2 = \theta + 1$), we have

$$\begin{aligned} a \cdot b =& a(0)b(0) + a(1)b(1) + \theta \cdot (a(0)b(1) + a(1)b(0) + a(1)b(1)) \\ &\implies (ab)(0) = a(0)b(0) + a(1)b(1). \end{aligned}$$

Now, assume that the parties reconstruct $b(1)$, which can be done using a single bit of communication per party from their shares $[\![b]\!]^4 = [\![b(0)]\!]^2 + \theta \cdot [\![b(1)]\!]^2$. Given $b(1)$, the parties can locally compute shares of $a(0)b(0)$ as follows:

$$[\![a(0)b(0)]\!]^2 = [\![ab]\!]^4(0) + b(1) \cdot [\![a]\!]^4(1).$$

Therefore, all parties output $([\![a(0)]\!]^2, [\![b(0)]\!]^2, [\![ab]\!]^4(0) + b(1) \cdot [\![a]\!]^4(1))$, which forms a valid Beaver triple over $\mathbb{F}_2$. Security is straightforward: the only communication between the parties is the reconstruction of $b(1)$, which is a uniformly random bit independent of $a(0), b(0)$. From there, one immediately gets an improved protocol in the preprocessing model: in the preprocessing phase, given one $\mathbb{F}_4$-Beaver triple for each AND gate of the circuit, the parties broadcast one bit per gate, and then locally derive the $\mathbb{F}_2$-Beaver triples. In the online phase, the parties run the standard GMW protocol. We refer the reader to the full version [7] of for the formal statement of this optimization.

2.4 An Improved Protocol from $\mathbb{F}_4$-OLEs for $N = 2$

In the setting of $N = 2$ parties, we obtain a much more efficient alternative: we observe that two parties can directly convert a single OLE over $\mathbb{F}_4$ into a Beaver triple over $\mathbb{F}_2$. (In contrast, recall that the standard approach requires two oblivious transfers for each triple.) We consider two parties, Alice and Bob, holding respectively $(a, [\![ab]\!]^4_A)$ and $(b, [\![ab]\!]^4_B)$ for a and $b \in \mathbb{F}_4$. We have

$$\begin{aligned} a \cdot b &= [\![ab]\!]^4_A(0) + [\![ab]\!]^4_B(0) + \theta \cdot ([\![ab]\!]^4_A(1) + [\![ab]\!]^4_B(1)) \\ &= (a(0)b(0) + a(1)b(1)) + \theta \cdot (a(0)b(1) + a(1)b(0) + a(1)b(1)), \end{aligned}$$

where θ is the primitive root of $X^2 + X + 1$. Considering only the $(a \cdot b)(0)$ term from the above equation (i.e., the parts not multiplied by θ), we get that

$$(a \cdot b)(0) = [\![ab]\!]^4_A(0) + [\![ab]\!]^4_B(0) = a(0)b(0) + a(1)b(1), \text{ and therefore,}$$

$$\underbrace{a(0)a(1) + [\![ab]\!]^4_A(0)}_{\text{known by } A} + \underbrace{b(0)b(1) + [\![ab]\!]^4_B(0)}_{\text{known by } B} = \underbrace{(a(0) + b(1))}_{\text{shared by } A,B} \cdot \underbrace{(a(1) + b(0))}_{\text{shared by } A,B}.$$

Above, the values $a(0)a(1) + [\![ab]\!]^4_A(0)$ (known by Alice) and $b(0)b(1) + [\![ab]\!]^4_B(0)$ (known by Bob) form additive shares of the product $(a(0) + b(1)) \cdot (a(1) + b(0))$, which Alice and Bob hold additive shares of. It is also easy to check that if the input is a random $\mathbb{F}_4$-OLE, the output is a random multiplication triple over $\mathbb{F}_2$. Therefore, following the local conversion procedure outlined above, Alice and Bob can transform a random $\mathbb{F}_4$-OLE instance into a random Beaver over $\mathbb{F}_2$ without having to communicate. We refer the reader to the full-version [7] for the formal statement of this optimization.

2.5 A Fast Programmable PCG for $\mathbb{F}_4$-OLEs

In light of the above observations, the only missing piece of the puzzle is an efficient way of generating a large number of $\mathbb{F}_4$-OLEs. In the $N > 2$ setting, if the OLEs are additionally programmable, the parties can afterward locally convert $N \cdot (N - 1)$ $\mathbb{F}_4$-OLE instances into an $\mathbb{F}_4$-Beaver triples.

Here, we build on the recent general programmable PCG construction of [8]. Because we are targeting OLEs over $\mathbb{F}_4$, we set the group $\mathbb{G}$ to $\mathbb{F}_3^n$, and the underlying group algebra becomes isomorphic to

$$\mathbb{F}_4[\mathbb{G}] \simeq \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1) \simeq \mathbb{F}_4^{3^n}.$$

Before delving into the optimizations we develop for their construction, we describe the high-level ideas and main building blocks behind the PCG construction of Bombar et al. [8] when instantiated over $\mathbb{F}_4$.

The PCG Construction of Bombar et al. As with previous constructions of PCGs [9,12], the construction of Bombar et al. uses Distributed Point Functions (DPF) [15,16,27] as a core building block. Informally, a DPF with domain $[D]$ allows a dealer to succinctly secret share a unit vector over $[D]$. The most efficient DPFs have shares of size roughly $\lambda \cdot \log D$ [16], for some security parameter λ, and the cost of decompressing the shares is dominated by D calls to a length-doubling pseudorandom generator.

Public Parameters. For a fixed *compression factor* c (typically a small constant, e.g., $c = 3$) and *noise parameter* t (e.g., $t = 27$), the public parameters contain a length-c vector $\mathbf{a}$ of n-variate polynomials.

Distributing PCG Seeds. In their construction, $\mathsf{PCG.Gen}$ does the following:

- it samples two length-c vectors $(\mathbf{e}_0, \mathbf{e}_1)$ of t-sparse polynomials over $\mathbb{F}_4[\mathbb{G}]$;
- outputs keys $(\mathsf{k}_0, \mathsf{k}_1)$ that contain $\mathbf{e}_0$ and $\mathbf{e}_1$, respectively, as well as *succinct* shares of $\mathbf{e}_0 \otimes \mathbf{e}_1$, encoded using a DPF.

The tensor product $\mathbf{e}_0 \otimes \mathbf{e}_1$ contains c^2 polynomials, each with at most t^2 nonzero coordinates. Hence, the vectors of coefficients of all polynomials in $\mathbf{e}_0 \otimes \mathbf{e}_1$ can be succinctly secret shared using $(ct)^2$ DPFs with domain 3^n, which requires roughly $(ct)^2 \cdot \lambda \log(3^n)$ bits using the state-of-the-art DPF constructions [15,16].[3]

Generating Correlations. To output a vector of OLE correlations, $\mathsf{PCG.Eval}$ proceeds as follows for party 0 (the evaluation for party 1 is similar):

– evaluate all the DPFs to obtain a secret share of $[\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_0$;
– set $x_0 \leftarrow \langle \mathbf{a}, \mathbf{e}_0 \rangle$ and $z_0 \leftarrow \langle \mathbf{a} \otimes \mathbf{a}, [\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_0 \rangle$; $\triangleright$ Note: $z_0 = [\![\langle \mathbf{a} \otimes \mathbf{a}, \mathbf{e}_0 \otimes \mathbf{e}_1 \rangle]\!]_0$
– using the isomorphism $\mathbb{F}_4[\mathbb{G}] \simeq \mathbb{F}_4^{3^n}$, project $(x_0, z_0) \in \mathbb{F}_4[\mathbb{G}]^2$ onto 3^n pairs (x_0^i, z_0^i) of elements of $\mathbb{F}_4$.

Above, the projection amounts to evaluating the multivariate polynomials over $\mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$ on the 3^n tuples of elements of $(\mathbb{F}_4^\times)^n$. Observe that

$$\begin{aligned} z_0 + z_1 &= \langle \mathbf{a} \otimes \mathbf{a}, [\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_0 \rangle + \langle \mathbf{a} \otimes \mathbf{a}, [\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_1 \rangle \\ &= \langle \mathbf{a} \otimes \mathbf{a}, \mathbf{e}_0 \otimes \mathbf{e}_1 \rangle = \langle \mathbf{a}, \mathbf{e}_0 \rangle \cdot \langle \mathbf{a}, \mathbf{e}_1 \rangle = x_0 \cdot x_1. \end{aligned}$$

Since the isomorphism preserves additions and multiplications, it follows that all pairs (x_0^i, z_0^i) and (x_1^i, z_1^i) form OLEs over $\mathbb{F}_4$. Security boils down to the Quasi-Abelian Syndrome Decoding assumption (QA-SD) [8], which states (informally) that given the random vector $\mathbf{a}$, the element $\langle \mathbf{a}, \mathbf{e} \rangle + e_0$ (where $(e_0, \mathbf{e})$ are formed by random sparse polynomials) is indistinguishable from a random element of $\mathbb{F}_4[\mathbb{G}]$.

We now describe several observations that we make about their construction and how these observations allow us to significantly optimize the concrete efficiency of the PCG. While simple in retrospect, these observations allow us to turn a theoretical construction into a concretely efficient PCG for $\mathbb{F}_4$-OLEs (see Sect. 5 for our implementation and evaluation).

Early Termination. The DPF construction of [16] generates shares of a unit vector using a construction *à la* GGM [28], generating a full binary tree of PRG evaluations starting from a root seed. The children of each node are computed by evaluating a length-doubling PRG on the node, and then adding some correction words. In this construction, each leaf of the tree is a λ-bit string (where typically $\lambda = 128$). In contrast, we wish to share unit vectors over $\mathbb{F}_4$. Hence, we can apply the *early termination* technique from [16] that shaves several levels of PRG expansions. With early termination, to obtain a $D = 2^d$-long vector over $\mathbb{F}_4$, we use a tree of depth $2D/\lambda = 2^{d-6}$ (using $\lambda = 128$) and parse each of the 128-bit leaves as a 64-tuple of $\mathbb{F}_4$-elements. This immediately yields a 64-fold runtime improvement for each of the DPFs required in the PCG construction.

We note that while other constructions share a similar blueprint to the construction of Bombar et al., and in particular also require evaluating many DPFs

[3] Using noise vector with a regular structure, the domain size of the DPFs can be reduced to $3^n/t$.

under-the-hood, this early termination technique does not apply to them. The reason is that in silent OT extension protocols [10–12,21,41], the DPFs are used to compress secret shares of $\Delta \cdot \mathbf{e}$, where Δ is a 128-bit element from a suitable extension field, and in the previous PCG construction of [14], the OLEs can only be generated over a large field $\mathbb{F}$ (chosen equal to $|\mathbb{F}| \approx 2^\lambda$ in their implementation). As such, early termination optimization appears to apply exclusively when specializing the PCG of [8] to work over small fields.

Using a Single Multi-evaluation Step. Computing $\langle \mathbf{a} \otimes \mathbf{a}, [\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_b \rangle$ (for $b = 0, 1$) requires c^2 polynomial multiplications. Fast polynomial multiplication is typically done using a multi-evaluation (i.e., an FFT) followed by a local product and an interpolation (i.e., an inverse FFT).

The above produces a single OLE over $\mathbb{F}_4[\mathbb{G}]$. When the end goal is to obtain OLEs over $\mathbb{F}_4$, the result is projected back onto $\mathbb{F}_4^{3^n}$ using a multi-evaluation. In this case, we show that we can reduce the sequence multi-evaluation $\rightarrow$ interpolation $\rightarrow$ multi-evaluation down to just a single multi-evaluation step. Concretely:

- Given that $\mathbf{a}$ is a random vector of polynomials (and part of the public parameters), it can directly be generated as c random length-3^n vectors over $\mathbb{F}_4^n$, corresponding to the vectors of the multi-evaluations of $\mathbf{a}$ over all n-tuples in $(\mathbb{F}_4^\times)^n$.
- The multi-evaluation of $\mathbf{a} \otimes \mathbf{a}$ can be computed once for all using pairwise products of elements of (the multi-evaluation of) $\mathbf{a}$, and included in the public parameters.
- Computing the multi-evaluation of $\langle \mathbf{a} \otimes \mathbf{a}, [\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_b \rangle$ amounts to computing the multi-evaluation of $[\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_b$ followed by component-wise inner products.

It follows that after expanding the shares $[\![\mathbf{e}_0 \otimes \mathbf{e}_1]\!]_b$, the cost of $\mathsf{PCG.Expand}$ is then dominated by c^2 instances of a multi-evaluation (i.e., an FFT). However, upon slightly closer inspection, we observe that it actually suffices to compute $c(c+1)/2$ FFTs (since the terms $e_0^i e_1^j$ and $e_0^j e_1^i$ share the same "coefficient" $a_i a_j$ in $\langle \mathbf{a} \otimes \mathbf{a}, \mathbf{e}_0 \otimes \mathbf{e}_1 \rangle$, hence the FFT can be evaluated on terms $e_0^i e_1^j + e_0^j e_1^i$ directly).

Blazing Fast FFT. Our next observation is that the FFT over the group algebra $\mathbb{F}_4[\mathbb{G}]$ is actually *extremely* efficient. Indeed, given a polynomial $P(X_1, \cdots, X_n)$, one can rewrite P as

$$P_0(X_1, \cdots, X_{n-1}) \ + \ X_n P_1(X_1, \cdots, X_{n-1}) \ + \ X_n^2 P_2(X_1, \cdots, X_{n-1}).$$

Let us denote $\mathsf{FFT}(P, n)$ the functionality that evaluates P on all n-tuples over $(\mathbb{F}_4^\times)^n$, and outputs a multi-evaluation vector $\mathbf{v}$. By the above formula, computing $\mathsf{FFT}(P, n)$ reduces to

- computing $\mathbf{v}_i \leftarrow \mathsf{FFT}(P_i, n-1)$ for each $i \in \{0, 1, 2\}$, and
- setting $\mathbf{v} \leftarrow (\mathbf{v}_0 + \mathbf{v}_1 + \mathbf{v}_2 \ || \ \mathbf{v}_0 + \theta \mathbf{v}_1 + (\theta+1)\mathbf{v}_2 \ || \ \mathbf{v}_0 + (\theta+1)\mathbf{v}_1 + \theta \mathbf{v}_2)$.

Denoting $C(n)$ the cost of running $\mathsf{FFT}(P, n)$, we therefore have $C(n) = 3 \cdot C(n-1) + \ell \cdot 3^{n-1}$, where ℓ denotes the number of vector operations (naïvely, 6 additions of vectors and 4 scalar-vector products—but some additions and products can be reused). This yields a cost of $C(n) = n \cdot \ell \cdot 3^{n-1}$, where all operations are very fast: either additions of $\mathbb{F}_4$-vectors or multiplications by θ. Looking ahead, our implementation and evaluation (Sect. 5) confirm that, even with the straightforward recursive algorithm, the FFT results in minimal overhead compared to the cost of the DPFs.[4]

Stepping Back: Comparison with Silent OT Extension. To give an intuition about the efficiency of this construction, we provide a brief comparison with constructions of silent OT extension. In short, to get (say) 3^n OTs, these constructions run $c \cdot t$ DPFs on a domain of size $3^n/t$, followed by a multiplication with a compressive mapping. In the most efficient silent OT extension protocol to date [41], this compressive mapping requires computing $21 \cdot c \cdot 3^n$ XORs, followed by 3^n XORs of random size-21 subsets of the bits of the resulting vector. Due to the overhead of many random memory accesses, the cost of computing this mapping dominates the overall runtime. In contrast, we need $(c \cdot t)^2$ DPFs with domain size $3^n/t$, but get a $64\times$ speedup from the early termination optimization. The cost of our DPFs should be essentially on par with that of [41]. However, the FFT cost in our construction is largely dominated by the cost of the DPFs. Therefore, we expect (and this is confirmed by our implementation) that this PCG should produce $\mathbb{F}_4$-OLEs at a much faster rate compared with the rate at which [41] produces OTs. In the two-party setting, when the goal is to generate Beaver triples over $\mathbb{F}_2$, we get an additional $2\times$ speedup from the technique of Sect. 2.4, as we generate one triple from *one* $\mathbb{F}_4$-OLE (whereas [41] requires two OTs). We provide an optimized implementation of our scheme and evaluate how it compares to previous works in Sect. 5. Our implementation is about $6\times$ faster than the state of the art [41].

2.6 Distributed Seed Generation

So far, we have only discussed the cost of expanding the PCG keys $(\mathsf{k}_0, \mathsf{k}_1)$. To obtain a full-fledged secure computation protocol, we need an efficient way for the parties to securely evaluate $\mathsf{PCG.Gen}$ procedure in a distributed fashion. In the following, as in all previous works on PCGs [8,10–14,20,21], we assume that the noise follows a regular distribution. That is, a noise vector $\mathbf{e}$ is a vector of c polynomials $(e^1, \cdots e^c)$, where each polynomial e^i is *regular*: its coordinates are divided into t block of (approximately) equal length $3^n/t$, and it has a single nonzero coefficient in each block. For any integer h, let $[h]$ denote the set $\{1, \cdots, h\}$. The previous work of [14] outlined the following methodology to securely distribute PCG seeds for generating D OLEs (in our context, $D = 3^n$):

- **Sampling the noise vectors.** Each party P_b generates its noise vector $\mathbf{e}_b$ locally, by sampling c t-sparse regular polynomials. We write $\mathbf{e}_b =$

[4] Our implementation also exploits vectorized operations to perform a batch of multiple FFTs for essentially the cost of one, which further reduces the impact of FFTs on the overall runtime.

$(e_b^1, \cdots, e_b^c)$. For each $i \in [c]$, we let $(\mathsf{p}_{b,1}^i, \cdots, \mathsf{p}_{b,t}^i) \in [3^n/t]^t$ denote the t positions of the nonzero entries in e_b^i, and $(v_{b,1}^i, \cdots, v_{b,t}^i) \in \mathbb{F}_4^t$ denote the value of these nonzero coefficients.

– **Sharing the positions and values.** For every $i_0, i_1 \in [c]$, for every $j_0, j_1 \in [t]$, the parties run a distributed protocol with respective inputs $\mathsf{p}_{0,j_0}^{i_0}$ and $\mathsf{p}_{1,j_1}^{i_1}$ (i.e., the position of the j_0-th and j_1-th nonzero coefficients in $e_0^{i_0}$ and $e_1^{i_1}$, respectively) which securely computes bitwise shares of the $(j_0 + j_1)$-th nonzero coefficient of $e_0^{i_0} e_1^{i_1}$. In parallel, they also run a distributed protocol with respective inputs $v_{0,j_0}^{i_0}$ and $v_{1,j_1}^{i_1}$ (the corresponding values of the nonzero coefficients) and securely compute bitwise shares of $v_{0,j_0}^{i_0} \cdot v_{1,j_1}^{i_1}$ (the value of the $(j_0 + j_1)$-th nonzero coefficient of $e_0^{i_0} e_1^{i_1}$).
– **Distributing the DPF keys.** For every $i_0, i_1 \in [c]$, for every $j_0, j_1 \in [t]$, the parties run the Doerner-shelat protocol [24] with their bitwise shares of the position and value to securely obtain DPF keys forming succinct shares of the point function $f_{\alpha,\beta}$ which evaluates to $\beta := v_{0,j_0}^{i_0} \cdot v_{1,j_1}^{i_1}$ on the index α of the $(j_0 + j_1)$-th nonzero coefficient of $e_0^{i_0} e_1^{i_1}$, and to 0 on all other inputs.

Communication-wise, the Doerner-shelat protocol requires $2 \cdot \log(D/t)$ oblivious transfers for each DPF, for a total of $2(ct)^2 \log(D/t)$ oblivious transfers. Distributing the shares of the coefficients $v_{0,j_0}^{i_0} \cdot v_{1,j_1}^{i_1}$ is relatively straightforward: it involves two OLEs over $\mathbb{F}_4$ for each of the (ct^2) coefficients. As in [14], these OLEs can be obtained at a minimal cost by running the PCG in a "bootstrapping mode": whenever two parties use the PCG to generate D $\mathbb{F}_4$-OLEs, they can instead use a marginally larger instance to generate $D + (ct)^2$ $\mathbb{F}_4$-OLE, and store the $(ct)^2$ extra OLEs for use in the next distributed PCG seed generation.

In the work of Boyle et al. [14], an important overhead comes from the $(ct)^2$ instances of a distributed protocol to generate bitwise shares of the noise positions: each such instance requires securely running a Boolean adder to compute, from the bit decomposition of $\mathsf{p}_{0,j_0}^{i_0}$ and $\mathsf{p}_{1,j_1}^{i_1}$, the bit decomposition of the position of the corresponding entry in $e_0^{i_0} e_1^{i_1}$. In the construction of [14], this contributes to a large portion of the (communication and computation) overhead of the seed distribution procedure: about half of the communication, computation, and rounds of the full protocol.

An Improved Seed Distribution from Ternary DPFs. We now introduce an optimization that removes the need to distribute shares of noise positions altogether by working *directly* in the ternary basis. Our improved protocol is tailored to the setting of noise vectors with components over $\mathbb{F}_4[\mathbb{G}] = \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$. Observe that every monomial over $\mathbb{F}_4[\mathbb{G}]$ can be written as $\mathbf{X}^{\mathsf{p}} := \prod_{i=1}^n X_i^{p_i}$, where $\mathsf{p} = (p_1, \ldots, p_n) \in \mathbb{F}_3^n$. Therefore, we can uniquely identify the position of the coefficient c_{p} of a monomial $\mathbf{X}^{\mathsf{p}}$ with the $\mathbb{F}_3$-vector $\mathsf{p} \in \mathbb{F}_3^n$. Now, consider the product of two polynomials e_0, e_1 known by P_0 and P_1, respectively. Let $\mathsf{p}_0 \in \mathbb{F}_3^n$ be the position of a nonzero entry in e_0, and $\mathsf{p}_1 \in \mathbb{F}_3^n$ be the position of a nonzero entry in e_1. Then, the corresponding nonzero entry in $e_0 \cdot e_1$ is the coefficient of the monomial $\mathbf{X}^{\mathsf{p}_0} \cdot \mathbf{X}^{\mathsf{p}_1} = \mathbf{X}^{\mathsf{p}_0 + \mathsf{p}_1 \bmod 3}$.

That is, the corresponding nonzero position in e_0e_1 is exactly $\mathsf{p}_0 + \mathsf{p}_1$ (where the sum is taken modulo 3). In other words, the two parties *already hold shares* of the noise position in e_0e_1—but over the ternary basis!

Unfortunately, the Doerner-shelat protocol requires the parties to hold binary shares of the position, because its binary decomposition corresponds to the path from the root to the leaf in the (binary) GGM tree underlying the DPF construction of [15,16]. To remedy this situation, we modify the underlying DPF construction to use a *ternary tree*. That is, the full tree is obtained by computing the three children of a node by evaluating a length-tripling PRG $G : \{0,1\}^\lambda \rightarrow \{0,1\}^{3\lambda}$ on the node value. Adapting the DPF construction of [15,16] to this setting is relatively simple (though the security analysis becomes slightly more tedious, especially when adapting the Doerner-shelat protocol to work over a ternary basis), and requires increasing the number of correction words from 1 to 3 per level of the tree.[5] With this change, the path to a leaf is given directly by the leaf position written as a $\mathbb{F}_3$-vector. To securely generate the keys of this modified DPF, we adapt the Doerner-shelat protocol. Our adaptation requires two 1-out-of-3 oblivious transfers per level (instead of two 1-out-of-2 OTs as in [24]), for the $\log_3(D/t)$ levels of the ternary DPF tree. In summary, we obtain a distributed seed generation protocol with the following pros-and-cons when compared to the original approach of [14]:

+ The parties "natively" hold shares of the nonzero positions and do not have to run a secure protocol to compute them. In the protocol of Boyle et al. [14], this step required $2(ct)^2 \cdot \log(D/t)$ oblivious transfers in $\log(D/t)$ rounds (i.e., half of the total number of rounds and OTs).
– The modified Doerner-shelat requires $2(ct)^2 \log_3(D/t)$ 1-out-of-3 OTs of 3λ-bit strings instead of $2(ct)^2 \log_2(D/t)$ 1-out-of-2 OTs of 2λ-bit strings, which represents slightly more communication and computation.
– Due to the use of a ternary DPF, which has more correction words, the PCG seed size is slightly increased, by a factor ≈ 1.5.
+ Expanding the PCG seeds becomes about 20% faster because the total number of PRG evaluations is reduced when computing a full ternary tree compared to a full binary tree with a similar number of leaves.
+ The number of rounds of the Doerner-shelat protocol is also reduced, from $\log_2(D/t)$ to $\log_3(D/t)$, by having a more shallow tree.

2.7 Concrete Cryptanalysis of $\mathbb{F}_4$OLEAGE

The security of $\mathbb{F}_4$OLEAGE is based on the QA-SD assumption, as explained in Sect. 2.2. We complement our construction with a thorough study of the security of QA-SD over small finite fields. Toward this end, we introduce a new combination of cryptanalytic techniques tailored to this setting, from which we

[5] Unfortunately, in the ternary tree construction, using the optimization described in [16] for removing one extra correction word does not immediately apply. We leave open the problem of finding a similar optimization in the ternary case.

derive improved attacks, and contribute a Sage script that computes the estimated bit security for any target instance. We believe that our analysis significantly improves our understanding of the concrete security of QA-SD, which had received little attention.[6] In the full version [7], we provide a high level overview of our analysis of QA-SD and of our new attack, tailored to the specificities of $\mathbb{F}_4$OLEAGE. To briefly summarize our conclusions: using our analysis and our Sage script, we estimate that taking $c = 3, t = 27$ offers about 128 bits of security, and taking $c = 4, t = 27$ provides a significant security margin (in both cases, the script is quite conservative on the power afforded to the adversary).

3 Preliminaries

Notations. We use $\mathbb{F}_4$ to denote the Galois field of order 4. For $\mathbb{G}$ an Abelian group, we denote by $\mathbb{F}_q[\mathbb{G}]$ the corresponding group algebra. For an integer n and a polynomial $f \in \mathbb{F}_q[\mathbb{G}]$ with n variables, $\mathsf{Eval}_n(f)$ denote the full evaluations of f over $(\mathbb{F}_q^\times)^n$. We let $[N]$ denote the set $\{1, 2, \ldots, N\}$. Divisibility is denoted as $a \mid b$, to mean a divides b. The number of elements in a list L is denoted as $|L|$. For a vector $\mathbf{e} \in \mathbb{F}_q^n$, we denote by $w_H(\mathbf{e})$ its Hamming weight. More generally, the Hamming weight of an element of a finite-dimensional $\mathbb{F}_q$-algebra $\mathcal{R}$ is the weight of the vector formed by its coefficients in some basis (in general, $\mathcal{R} = \mathbb{F}_q[\mathbb{G}]$ and we consider a basis given by an arbitrary ordering of the elements of $\mathbb{G}$.[7]). For an integer t, we denote by $\mathcal{R}_t$ the subset of elements of Hamming weight t. We denote by $\mathsf{poly}(\cdot)$ any polynomial and by $\mathsf{negl}(\cdot)$ any negligible function. We use $x \leftarrow_R S$ to denote a uniformly random sample drawn from S, and $x \leftarrow \mathcal{A}$ to denote assignment from a possibly randomized algorithm Adv. We use $x := y$ to denote the initialization of a value x to the value of y. We use $A \simeq B$ to indicate that two sets are isomorphic. By an *efficient* algorithm $\mathcal{A}$ we mean that Adv is modeled by a (possibly non-uniform) Turing Machine that runs in probabilistic polynomial time. We write $D_0 \approx_c D_1$ to mean that two distributions D_0 and D_1 are *computationally* indistinguishable to all efficient distinguishers $\mathcal{D}$ and $D_0 \approx_s D_1$ to mean that D_0 and D_1 are *statistically* indistinguishable.

Vectors and Tensor Products. We denote vectors using bold lowercase letters. For two vector $\mathbf{u} = (u_1, \ldots, u_t), \mathbf{v} = (v_1, \ldots, v_t) \in R^t$ for some ring R, their tensor product $\mathbf{u} \otimes \mathbf{v}$ is defined by $\mathbf{u} \otimes \mathbf{v} = (u_i \cdot v_j)_{i,j \leq t} = (v_1 \cdot \mathbf{u}, \ldots, v_t \cdot \mathbf{u})$ and we denote by $\langle \mathbf{u}, \mathbf{v} \rangle$ their inner product. Similarly, we write $\mathbf{u} \boxplus \mathbf{v}$ to denote the outer sum of a vector, equal to $\mathbf{u} \boxplus \mathbf{v} = (u_i + v_j)_{i,j \leq t} = (v_1 + \mathbf{u}, \ldots, v_t + \mathbf{u})$. We let $\mathbf{u}[i]$ denote the value of index i in $\mathbf{u}$.

[6] In contrast, the *asymptotic* security of QA-SD is much better studied: Attacking this assumption has been a long standing open problem in coding theory for more than fifty years (see e.g. Research Problem 16.10.5 p. 382 of [32]). Moreover, it was proven in [8] that it resists all attacks from the linear test framework—which captures most known attacks against syndrome decoding and its variants—and that it admits a search-to-decision reduction.

[7] The Hamming weight does not depend on the ordering of the elements of $\mathbb{G}$.

3.1 Function Secret Sharing

Function secret sharing (FSS), introduced in [15,16], allows a dealer to succinctly secret share a function with two parties. An FSS scheme splits a secret function $f : \mathcal{D} \rightarrow \mathbb{G}$, where $\mathbb{G}$ is some Abelian group into *keys* K_0, K_1 that can be used by party $\sigma \in \{0, 1\}$ to evaluate the function on an input $x \in \mathcal{D}$ and obtain the share $[\![f(x)]\!]_\sigma$ of the result. We focus on FSS for *point functions* which are known as Distributed Point Functions (DPFs).

Distributed Point Functions. Let $\mathcal{D}$ be an input domain and $\mathbb{G}$ be an Abelian group. A *point function* $P_{\alpha,\beta} : \mathcal{D} \rightarrow \mathbb{G}$ is a function that evaluates to message $\beta \in \mathbb{G}$ on a single input $\alpha \in \mathcal{D}$, and evaluates to $0 \in \mathbb{G}$ on all other inputs $x \neq \alpha \in \mathcal{D}$. A *distributed* point function (Definition 1) is a point function that is encoded into a pair of keys. Each key can be used to obtain an additive *secret-share* of the point function $P_\alpha(x)$, for any input $x \in \mathcal{D}$.

Definition 1 (Distributed Point Function (DPF) [16,27]). *Let λ be the security parameter, $\mathcal{D}$ be an input domain, and $\mathbb{G}$ be an Abelian group. A* DPF *scheme (with a full-domain evaluation procedure) consists of a tuple of efficient algorithms* $\mathsf{DPF} = (\mathsf{Gen}, \mathsf{FullEval})$ *with the following syntax.*

- $\mathsf{DPF.Gen}(1^\lambda, 1^n, \alpha, \beta) \rightarrow (K_0, K_1)$. *Takes as input a security parameter, a domain size n, and index $\alpha \in \mathcal{D}$ and a payload $\beta \in \mathbb{G}$. Outputs two evaluation keys K_0 and K_1.*
- $\mathsf{DPF.FullEval}(\sigma, K_\sigma) \rightarrow \mathbf{v}_\sigma$. *Takes as input the party index σ and an evaluation key K_σ. Outputs a vector $\mathbf{v}_\sigma$.*

These algorithms must satisfy correctness, security, and efficiency:

Correctness. *A DPF is said to be* correct *if for all $\alpha \in \mathcal{D}$, all $\beta \in \mathbb{G}$, and all pairs of keys generated according to* $\mathsf{DPF.Gen}(1^\lambda, 1^n, \alpha, \beta)$, *the sum of the individual outputs from* $\mathsf{DPF.FullEval}$ *result in the one-hot basis vector scaled by the message β,*

$$\Pr\left[\ \mathsf{FullEval}(0, K_0) + \mathsf{FullEval}(1, K_1) = \beta \cdot \mathbf{e}_{\boldsymbol{\alpha}}\ \right] = 1,$$

where $\mathbf{e}_{\boldsymbol{\alpha}} \in \mathbb{G}^{|\mathcal{D}|}$ is the α-th basis vector.

Security. *A DPF is said to be* secure *if each individual evaluation key output by* $\mathsf{DPF.Gen}$ *leaks nothing about (α, β) to a computationally bounded adversary. Formally, there exists an efficient simulator $\mathcal{S}$ such that $\{K_\sigma\} \approx_c \mathcal{S}(1^\lambda, 1^n, \sigma)$, where $\approx_c$ denotes the computational indistinguishability of distributions.*

Efficiency. *A DPF is said to be* efficient *if the size of each key is sublinear in the domain size. That is, for all $\sigma \in \{0, 1\}$, $|K_\sigma| = |\mathcal{D}|^\epsilon$ for some $\epsilon < 1$.*

FSS for the Sum of Point Functions. We let SPFSS be an FSS scheme for the class of *sums of point functions*: Functions of the form $f(x) = \sum_i f_{s_i,y_i}(x)$, where each $f_{s_i,y_i}(\cdot)$ evaluates to y_i on s_i, and to 0 everywhere else. As in previous works, we will use efficient constructions of SPFSS in our constructions of PCGs.

3.2 Quasi-Abelian Syndrome Decoding (QASD)

We recall about QASD assumption and its notation. More details on the QA-SD assumption is provided in the full version [7]. A finite Abelian group is a direct product of cyclic group: $\mathbb{G} \simeq \mathbb{Z}/d_1\mathbb{Z} \times \cdots \times \mathbb{Z}/d_r\mathbb{Z}$ where the d_i's can be equal. Then, the group algebra $\mathbb{F}_q[\mathbb{G}]$ admits an explicit description as some particular multivariate polynomial ring:

$$\mathbb{F}_q[\mathbb{G}] \simeq \mathbb{F}_q[X_1, \ldots, X_r]/(X_1^{d_1} - 1, \ldots, X_r^{d_r} - 1),$$

where the isomorphism is given by $(k_1, \ldots, k_r) \mapsto X_1^{k_1} \cdots X_r^{k_r}$, and extended by linearity.

Definition 2 **(QA-SD$(q, c, t, \mathbb{G})$).** *Let $\mathbb{G}$ be a finite Abelian group, $\mathbb{F}_q[\mathbb{G}]$ its algebra with coefficients in the finite field $\mathbb{F}_q$, and let $c \geq 2$ be some constant integer called the compression factor. Given a target Hamming weight $t \in \{1, \ldots, |\mathbb{G}|\}$ and a probability distribution Φ_t which outputs elements $x \in \mathbb{F}_q[\mathbb{G}]$ such that $\mathbb{E}(w_H(x)) = t$, the* Quasi-Abelian Syndrome Decoding *problem asks to distinguish, with a non-negligible advantage, between the distributions:*

$$\mathcal{D}_0 : \quad \left((a^{(i)})_{i \in \{1,\ldots,c-1\}}, u\right) \qquad \text{where } a^{(i)}, u \xleftarrow{\$} \mathbb{F}_q[G]$$

$$\mathcal{D}_1 : \left((a^{(i)})_{i \in \{1,\ldots,c-1\}}, \sum_{i=1}^{c-1} a^{(i)} e_i + e_0\right) \quad \text{where } a^{(i)} \xleftarrow{\$} \mathbb{F}_q[G] \text{ and } e_i \xleftarrow{\$} \Phi_t.$$

We say that the QA-SD$(q, c, t, \mathbb{G})$ assumption holds when this problem is hard for every non-uniform polynomial time distinguisher.

4 A Fast PCG for $\mathbb{F}_4$-OLEs

4.1 PCGs over $\mathbb{F}_4$ from the QA-SD Assumption

In [8], the authors point out that their QA-SD$_{\mathsf{OLE}}$ construction is the first to produce a large number of OLE correlations over $\mathbb{F}_q$, for any $q \geq 3$. They propose using $\mathbb{G} = \prod_{i=1}^{n} \mathbb{Z}/(q-1)\mathbb{Z}$, $q \geq 3$. The direct consequence of this is that $\mathbb{F}_q[\mathbb{G}] \simeq \mathbb{F}_q[X_1, \ldots, X_n]/(X_1^{q-1} - 1, \ldots, X_n^{q-1} - 1) \simeq \prod_{i=1}^{D} \mathbb{F}_q$, where the last isomorphism equivalence comes from the Chinese Remainder Theorem. Above, $D = (q-1)^n$ is the number of elements in the group, and the number of OLE's we can get over $\mathbb{F}_q$ by applying this isomorphism. Looking closely, we instantiate our particular PCG over $\mathcal{R} \simeq \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$, by setting $q = 4$. At the end of the protocol QA-SD$_{\mathsf{OLE}}$, the parties obtain one OLE over $\mathcal{R}$. Let us denote (x_σ, z_σ) the output of party σ. To obtain many OLE's over $\mathbb{F}_4$, the parties have to evaluate $x_\sigma, z_\sigma \in \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$ over the full domain $(\mathbb{F}_4^\times)^n$. The standard approach is to use a Fast Fourier Evaluation to efficiently obtain this result. Here, we remark that, in our group algebra, fast multiplication *also* requires FFT, first in a multi-evaluation form, and then in the interpolation form. Therefore, doing the interpolation again is wasteful as in the end we will evaluate again after interpolating. As such, we can avoid the intermediate steps of multi-evaluation-then-interpolation and work directly with the evaluations, without coming back to $\mathcal{R}$. That is, we do not construct the polynomials x_σ, z_σ over $\mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$ but instead, we focus directly on the polynomials evaluations.

Let $\mathsf{Eval}_n(f) = \{f(x_1, \ldots, x_n), (x_1, \ldots, x_n) \in \{1, \theta, \theta + 1\}^n\}$ be the set of all the possible evaluations. Instead of giving the parties the description of the coefficients of the polynomials $a_i \in \mathbf{a}$, we can give them the vectors of all the evaluations of all the polynomials, that is giving them $\mathsf{Eval}_n(a_i)$, for all i. Because we can write $x_\sigma = e_\sigma^0 + e_\sigma^1 a_1 + \cdots + e_\sigma^{c-1} a_{c-1}$, it follows that all the evaluations of x_σ can be obtained from $\mathsf{Eval}_n(e_\sigma^i)$ and $\mathsf{Eval}_n(a_i)$. All that remains is to evaluate the e_σ^i polynomials. They are sparse polynomials, and therefore their evaluations can be computed very efficiently i.e., if the polynomials have t nonzero coefficients, then the cost of the evaluation is linear in $t \cdot 3^n$. As a result, we can obtain $\mathsf{Eval}_n(x_\sigma)$ for a cost linear in 3^n.

The computation of $\mathsf{Eval}_n(z_\sigma)$ is a little trickier. As mentioned above, $x_0 \cdot x_1$ can be seen as a function of degree 2 in $(\mathbf{e_0}, \mathbf{e_1})$, with constant coefficients depending solely from $\mathbf{a} \otimes \mathbf{a}$. Because $\mathsf{Eval}_n(a_i)$ is already given to the parties, the evaluation of the coefficient from $\mathbf{a} \otimes \mathbf{a}$ can be obtained using only c^2 multiplications. It remains to compute the evaluations of the additive shares of the polynomials $e_0^i \cdot e_1^j$. There are c^2 such polynomials shared among the parties, and we can view each share as a random polynomial. Therefore, each party has to compute the evaluation of c^2 random polynomials. This is a crucial part of the scheme and we devote the next section to it. Figure 1 represents the PCG framework tailored to our setting, its correctness and security are implied by [8] (see the full version of this work [7] for details).

Remark 3. Let $t = 3^k$ be a power of 3, and let $\mathcal{R} = \mathbb{F}_4[\mathbb{G}] = \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$. Let $\mathbf{e_0}$ and $\mathbf{e_1}$ be sampled from a t-regular noise distribution over $\mathcal{R}$. In other words, the coordinates of $\mathbf{e_i}$ can be divided into t consecutive blocks $B_0, \ldots, B_{t-1}$ of size $3^n/t$, each block having a single nonzero coordinate. More precisely, considering the lexicographic ordering of the monomials, and since $t = 3^k$, block B_i is formed by all monomials $\mathbf{X^p}$ such that the first k coordinates of $\mathbf{p}$ represent the ternary decomposition of the integer i (over k trits). For example, if $n = 4$ and $t = 9$, the $3^4 = 81$ monomials are split into 9 blocks $B_0, \ldots, B_8$ of size 9, and a monomial $\mathbf{X^p}$ lies in B_6 if and only if $\mathbf{p}$ is of the form $(2, 0, \star, \star)$ with $\star \in \{0, 1, 2\}$, where $[2\|0]$ is the ternary decomposition of the integer 6.

We now show that the product $\mathbf{e} = \mathbf{e}_0 \cdot \mathbf{e}_1$ has at most t nonzero monomials in each block.[8] Indeed, let $i \in \{0, \ldots, 3^k - 1\}$ and let $\mathbf{X^p}$ be a monomial appearing in $\mathbf{e}$ with a nonzero coefficient. In particular, the first k entries of $\mathbf{p}$ can be parsed as the ternary decomposition of i, which we denote by $[i]_3$. It is clear that $\mathbf{X^p}$ is

[8] This crucially relies on the fact that since t is a power of 3, we can uniquely identify the block corresponding to a given monomial by looking at the first k entries of its exponent. When t is not a power of 3, this is not true anymore.

of the form $\mathbf{X}^{\mathbf{p}_0+\mathbf{p}_1}$ where $\mathbf{p}_0$ (resp. $\mathbf{p}_1$) identifies one of the t nonzero monomials in $\mathbf{e}_0$ (resp. $\mathbf{e}_1$), and the sum is taken modulo 3 component-wise. In particular, there are at most t^2 such monomials, and for each nonzero monomial $\mathbf{X}^{\mathbf{p}_0}$ of $\mathbf{e}_0$, with first k entries $[i_0]_3$, there corresponds *at most one* nonzero monomial in $\mathbf{e}_1$ contributing to $\mathbf{X}^{\mathbf{p}}$, namely $\mathbf{X}^{\mathbf{p}-\mathbf{p}_0}$.[9] In other words, the monomial $\mathbf{X}^{\mathbf{p}}$ can be produced by *at most* t possible pairs of monomials $(\mathbf{X}^{\mathbf{p}_0}, \mathbf{X}^{\mathbf{p}_1})$, whose first k entries are $([i_0]_3, [i]_3 - [i_0]_3)$, with i_0 ranging over $\{0, \ldots, t-1\}$.

Example. Let $n = 3$ and $t = 3$. Set $\mathbf{e}_0 := X_3^2 + X_1X_2X_3 + X_1^2$ (which corresponds to positions $(0,0,2), (1,1,1),$ and $(2,0,0)$) and $\mathbf{e}_1 := 1 + X_1 + X_1^2$ (which corresponds to positions $(0,0,0), (1,0,0)$, and $(2,0,0)$). Then,

$$\mathbf{e}_0 \cdot \mathbf{e}_1 = \underbrace{(1 + X_3^2 + X_2X_3)}_{\in B_0} + \underbrace{(X_1 + X_1X_3^2 + X_1X_2X_3)}_{\in B_1} + \underbrace{(X_1^2 + X_1^2X_3^2 + X_1^2X_2X_3)}_{\in B_2}.$$

Proposition 4. *Let $\mathcal{R} = \mathbb{F}_4[\mathbb{G}] = \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$ where $\mathbb{G} = \prod_{i=1}^n \mathbb{Z}/3\mathbb{Z}$ is an Abelian group. Assume that* SPFSS *is a secure* FSS *scheme for sums of point functions and that the* QA-SD$(q, c, t, \mathbb{G})$ *assumption holds for regular noise distribution. Then there exists a generic scheme to construct a* PCG *to produce one* OLE *correlation (described on Fig. 1). If the* SPFSS *is based on a* PRG $: \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ *via the* PRG*-based construction from [16], we obtain:*

- *Each party's seed has maximum size around:* $(c \cdot t)^2 \cdot ((n \cdot \log(3) - \log t + 1) \cdot (\lambda + 2) + \lambda + 2) + c \cdot t \cdot (n \cdot \log(3) + 2)$ *bits.*
- *The computation of* Expand *can be done with at most* $\log(3) \cdot (2 + \lfloor (2)/\lambda \rfloor) \cdot n \cdot c^2 \cdot t$ PRG *operations, and* $O(n \cdot \log(3) \cdot c^2 \cdot 3^n)$ *operations in* $\mathbb{F}_4$.

4.2 Optimizing the FSS Evaluation via Early Termination

We remark that we can use a very simple trick that enables the parties to obtain the evaluation of their MPFSS shares 64 times faster than with the standard construction (*and* at a slight reduction in communication). The trick comes from the fact that the standard construction of the DPF based on the GGM tree implies that each leaf is of size $\lambda = 128$ bits. It was pointed out in [16] that we can consider *early termination* in the case of small outputs. In our case, we would like a single leaf to encode a value in $\mathbb{F}_4$. This only requires 2 bits instead of the 128 bits we get as output, making the naïve evaluation "waste" 126 bits of the output. Instead, we can avoid wasting computation by truncating the tree 6 levels earlier and setting the value of the new 128-bit leaf on the special path to encode a unit vector consisting of zeroes except on the exact 2 bits where it equals to the correct value of $\mathbb{F}_4$ element. This essentially involves "hard-coding" the end of the path into the leaf directly, as illustrated in Sect. 4.2. Using this idea, we reduce the computational cost of evaluating the DPF by $64\times$ and reduce the communication costs (key size of the DPF) by roughly $6 \cdot 128$ bits [16]. This simple trick was initially introduced in the context of PIR applications [16], but could

[9] Note that the corresponding monomial $\mathbf{X}^{\mathbf{p}_1}$ might not appear in $\mathbf{e}_1$.

Specific construction of QA-SD$_{\mathsf{OLE}}$ for $\mathbb{F}_4$OLEAGE

Parameters: Noise weight $t = t(\lambda)$, compression factor c, ring $\mathcal{R} = \mathbb{F}_4[X_1, \ldots, X_n]/(X_1^3 - 1, \ldots, X_n^3 - 1)$. A SPFSS scheme SPFSS = (SPFSS.Gen, SPFSS.FullEval) for sums of t^2 point functions, with domain $[0 \ldots 3^n)$ and range $\mathbb{F}_4$.

Public Input: $c-1$ vectors of length 3^n over $\mathbb{F}_4$, corresponding to the result of $\mathsf{Eval}_n(a_i)$, for uniformly random $a_1, \cdots, a_{c-1} \in \mathcal{R}$, therefore the full evaluation of the c elements a_i.

```
PCG.Gen($1^\lambda$):
1: foreach $\sigma \in \{0,1\}$, $i \in [0 \ldots c)$:
    1.1: Sample random $\mathbf{p}_{\boldsymbol{\sigma}}^{\boldsymbol{i}} \leftarrow \{(\mathbf{p}_{\boldsymbol{\sigma},\mathbf{1}}^{\boldsymbol{i}}, \ldots, \mathbf{p}_{\boldsymbol{\sigma},\boldsymbol{t}}^{\boldsymbol{i}}) \mid \mathbf{p}_{\boldsymbol{\sigma},\boldsymbol{j}}^{\boldsymbol{i}} \in \mathbb{F}_3^n\}$, and $\mathbf{v}_{\boldsymbol{\sigma}}^{\boldsymbol{i}} \leftarrow (\mathbb{F}_4^\times)^t$.
        ▷ [Optimization]: $\mathbf{p}_{\boldsymbol{\sigma}}^{\boldsymbol{i}}$ can be sampled from regular noise distribution.
2: foreach $i, j \in [0 \ldots c)$:
    2.1: Sample FSS keys $(K_0^{i,j}, K_1^{i,j}) \leftarrow$ SPFSS.Gen$(1^\lambda, 1^n, \mathbf{p}_{\mathbf{0}}^{\boldsymbol{i}} \boxplus \mathbf{p}_{\mathbf{1}}^{\boldsymbol{j}}, \mathbf{v}_{\mathbf{0}}^{\boldsymbol{i}} \otimes \mathbf{v}_{\mathbf{1}}^{\boldsymbol{j}})$.
        ▷ If using regular noise as an optimization, then
        ▷ SPFSS is for the sum of $t$ point functions with domain $[0, \ldots, 3^n/t)$.
3: Let $\mathsf{k}_\sigma = ((K_\sigma^{i,j})_{i,j \in [0 \ldots c)}, (\mathbf{p}_{\boldsymbol{\sigma}}^{\boldsymbol{i}}, \mathbf{v}_{\boldsymbol{\sigma}}^{\boldsymbol{i}})_{i \in [0 \ldots c)})$.
4: Output $(\mathsf{k}_0, \mathsf{k}_1)$.

PCG.Expand($\sigma$, $\mathrm{K}_\sigma$):
1: Parse $\mathsf{k}_\sigma$ as $((K_\sigma^{i,j})_{i,j \in [0 \ldots c)}, (\mathbf{p}_{\boldsymbol{\sigma}}^{\boldsymbol{i}}, \mathbf{v}_{\boldsymbol{\sigma}}^{\boldsymbol{i}})_{i \in [0 \ldots c)})$.
2: foreach $i \in [0 \ldots c)$:
    2.1: Define over $\mathbb{F}_4$ the polynomial:
         $$e_\sigma^i(X) = \sum_{j \in [0 \ldots t)} \mathbf{v}_{\boldsymbol{\sigma}}^{\boldsymbol{i}}[j] \cdot \mathbf{X}^{\mathbf{p}_{\boldsymbol{\sigma}}^{\boldsymbol{i}}[j]}.$$
    2.2: Compute $\mathsf{Eval}_n(e_\sigma^i)$.
3: Compute $x_\sigma = \langle \mathbf{a}, \mathbf{e}_{\boldsymbol{\sigma}} \rangle$, where $\mathbf{a} = (1, a_1, \cdots, a_{c-1}), \mathbf{e}_{\boldsymbol{\sigma}} = (e_\sigma^0, \cdots, e_\sigma^{c-1})$.
4: From $\mathsf{Eval}_n(e_\sigma^i)$ and $\mathsf{Eval}_n(a_i)$, compute $\mathsf{Eval}_n(x_\sigma)$.
5: foreach $i, j \in [0 \ldots c)$,
    5.1: Compute $u_{\sigma,i+cj} \leftarrow$ SPFSS.FullEval$(\sigma, K_\sigma^{i,j})$ and view it as a $c^2$ vector $\mathbf{u}_{\boldsymbol{\sigma}}$
         of elements in $\mathcal{R}$.
6: foreach $j \in [0 \ldots c^2)$:
    6.1: Compute $\mathsf{Eval}_n(u_{\sigma,j})$.
        ▷ [Optimization]: only need to perform $c(c+1)/2$ FFTs, see Section 2.5.
7: Compute $\mathsf{Eval}_n(z_\sigma)$, with $z_\sigma = \langle \mathbf{a} \otimes \mathbf{a}, \mathbf{u}_{\boldsymbol{\sigma}} \rangle$.
8: Output $(\mathsf{Eval}_n(x_\sigma), \mathsf{Eval}_n(z_\sigma))$.
```

Fig. 1. QA-SD-based PCG for OLE over $\mathcal{R}$ from evaluations of functions.

not be applied to prior PCG constructions until now since all PCG constructions (except for the recent PCG construction of Bombar et al. [8]) required the DPF output to be encode elements of a large field. Similarly, in silent OT extension protocols [10–12, 21, 41], which are also bottlenecked by DPF evaluations, this optimization could not be applied because there, the DPF is used to output "authenticated" shares of a (potentially small) field element with a (large) MAC, which requires the leaves to encode 128 bit output value (Fig. 2).

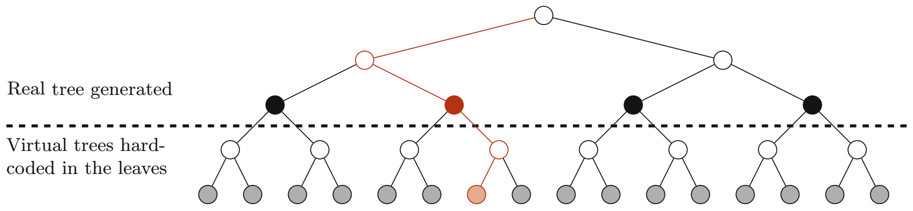


Fig. 2. Early termination example in the case we truncate only two steps earlier. Solid black nodes represent "zero" leaves, whereas solid red leaves can take on any value.

4.3 Fast Evaluation over $\mathbb{F}_4[X_1,\ldots,X_n]/(X_1^3-1,\ldots,X_n^3-1)$

The High-Level Idea. Given a polynomial P with n variables, the party wants to compute $\mathsf{Eval}_n(P)$, that is, to evaluate P over $\left(\mathbb{F}_4^\times\right)^n$ where $\mathbb{F}_4^\times = \{1, \theta, \theta+1\}$. Here, we adapt the standard divide-and-conquer style algorithm to our case, see for example the seminal work of [19]. Remark that

$$P(X_1,\ldots,X_n) = P_0(X_1,\ldots,X_{n-1}) + X_nP_1(X_1,\ldots,X_{n-1}) + X_n^2P_2(X_1,\ldots,X_{n-1}).$$

Instead of classically dividing our problem into 2 sub-problems, we divide it into 3 sub-problems. This is a ternary generalization of a standard FFT algorithm adapted to our case. Then,

$$\mathsf{Eval}_n(P) = \mathsf{Eval}_{n-1}(P_0) \cup X_n\mathsf{Eval}_{n-1}(P_1) \cup X_n^2\mathsf{Eval}_{n-1}(P_2). \tag{1}$$

Denote by $\mathcal{C}(\mathsf{Eval}_n(P))$ the number of operations carried out to obtain all the 3^n evaluations on the set $\mathbb{F}_4^\times$. Then we have $\mathcal{C}(\mathsf{Eval}_n(P)) = 3\mathcal{C}(\mathsf{Eval}_n(P)) + 2\cdot 3^n$, which leads us to $\mathcal{C}(\mathsf{Eval}_n(P)) = 4\cdot n\cdot 3^n$. The concrete number of additions or multiplications is $2\cdot n\cdot 3^n$. This quick back-of-the-envelope calculation captures the essence of the technique, even if it does not accurately count the cost of the various operations and does not take into account what is implemented in practice. We now turn to a concrete implementation of this idea:

Concrete Implementation. An element of $\mathbb{F}_4$ has a direct canonical representation using 2 bits. Given an element $x \in \mathbb{F}_4$, we write $x(0)$ and $x(1)$ to denote the $\mathbb{F}_2$-coefficients of x viewed as a polynomial over $\mathbb{F}_2[X]/(X^2+X+1)$; that is, $x = x(0) + \theta \cdot x(1)$. Using a given machine word of 64 bits we represent a vector of size 32 over $\mathbb{F}_4$, such that the even indexed bits are high order and the odd indexed bits are low order.

As stated before, we use a recursive algorithm to compute all the evaluations, displayed in algorithm Fig. 3. We considered using a non-recursive approach but no significant efficiency gains were observed, so we instead decided to use the recursive algorithm due to its conceptual simplicity.

Actual Cost of the Computation. A step in the algorithm of Fig. 3 is to evaluate a polynomial of degree 2, with coefficient in $\mathbb{F}_4$, for the values $\{1, \theta, \theta + 1\}$. Let the polynomial be $a + bX_i + cX_i^2$.

– in the case $X_i = 1$, then the evaluation of the polynomial becomes $a + b + c$.
– in the case $X_i = \theta$, the evaluation becomes $(a + c) + \theta \cdot (b + c)$.
– in the case $X_i = \theta + 1$, the evaluation becomes $(a + b) + \theta \cdot (b + c)$.

Note that we want to compute all the different evaluations, and therefore we can try to reduce the overall costs by reusing several of the intermediate calculations. We can obtain the three evaluations via the following steps: (1)compute $a + b, a + c, b + c$; (2) compute $\theta \cdot (b + c)$; (3) compute $a + b + c$; (4) Compute $(a + c) + \theta \cdot (b + c))$, and $(a + b) + \theta \cdot (b + c))$. Therefore, we count 12 classical bit-by-bit XOR over $\mathbb{F}_2$, and a multiplication by θ to obtain the three needed evaluations of the polynomial.

Fast-Evaluation algorithm

```
PARAMETERS: $n > 0$ an integer, $P \in \mathbb{F}_3[X_1, \cdots, X_n]/(X_1^3 - 1, \cdots, X_n^3 - 1)$, a polynomial with $n$ variables.
FastEval($n, P$):
1: if $n = 1$ then
    1.1: return $\{P(1), P(\theta), P(\theta + 1)\}$
2: else
    2.1: Write $P(X_1, \cdots, X_n) = P_0(X_1, \cdots, X_{n-1}) + X_n P_1(X_1, \cdots, X_{n-1}) + X_n^2 P_2(X_1, \cdots, X_{n-1})$.
    2.2: $S := \{\}$.
    2.3: $\forall i \in \{0, 1, 2\}, S_i \leftarrow$ FastEval($n - 1, P_i$)
    2.4: foreach $i \in [|S_0|]$:
        2.4.1: $f_j(X) := S_0[j] + S_1[j]X + S_2[j]X^2$.
        2.4.2: $S \leftarrow S \cup \{f_j(1), f_j(\theta), f_j(\theta + 1)\}$.
    2.5: return $S$.
```

Fig. 3. Fast evaluation of a polynomial in n variables.

4.3.1 Taking Advantage of the Computer Words. Today's processors offer XOR operations for machine words of size 64 bits. We take advantage of this parallelism to run multiple FFTs in parallel with a small overhead compared to running a single FFT. With 64-bit machine words, we can perform up to 32 FFT in parallel. We pack the c^2 FFTs required by our PCG as follows: we let each machine word contain a single coefficient of the same monomial for each of the c^2 polynomials that we are trying to compute. This saves a factor of c^2, at no extra cost.[10] Therefore, the cost of the evaluation of a single polynomial being of $16n \cdot 3^{n-1}$ XOR, the optimization entails the cost of obtaining the full evaluation of the c^2 polynomials to be $16\lceil c^2/64 \rceil n \cdot 3^{n-1}$.

5 Implementation and Evaluation

We implement $\mathbb{F}_4$OLEAGE in C (v15.0.0) as a library that consists of two main components: (1) an optimized implementation of the ternary DPF construction and (2) an implementation of the FFT over $\mathbb{F}_4$. The open-source code for our $\mathbb{F}_4$OLEAGE PCG benchmarks is available online.[11]

Implementation Details. Our DPF implementation takes advantage of the `AES-NI` instruction to implement a fast PRG G using fixed-key AES (from the OpenSSL library [39]) and the Davies-Meyer transform. We experimented with using the half-tree optimization of [30]. However, we observed a minimal performance gains (2–4%) from this optimization when applied to a ternary tree. This is because the half-tree optimization is tailored to the binary tree DPF construction where it can shave a larger fraction of total AES calls. We implement the recursive FFT over $\mathbb{F}_4$ described in Sect. 4.3 and perform the FFT in parallel by packing all the coefficients into one machine word (for our parameters, we will require 16 FFTs, so we can perform them in parallel using a `uint32` type for packing). While the FFT could possibly be optimized further using an iterative algorithm and taking advantage of AVX instructions, the simplicity of the recursive algorithm coupled with the parallel packing makes it sufficiently fast for $\mathbb{F}_4$OLEAGE. This is especially true given that the DPF evaluations end up being the dominant cost (roughly 70% of the total computation). We do not implement the distributed seed generation protocol given that it consists of black-box invocations of any one-out-of-three OT. However, we do estimate the concrete performance and communication costs of distributed seed generation by benchmarking the libOTe library on state-of-the-art OT protocols [42].

Benchmarks. We perform our benchmarks using AWS `c5.metal` (3.4GHz CPU) and `t2.large` instances. All experiments are averaged across ten trials and evaluated on a single core. To better see the overhead involved with each component, we start by benchmarking the SPFSS (sum of many DPFs) and FFT implementation separately and report the results in Tables 2 and 3. Concretely,

[10] In practice, using larger machine words has an impact by increasing stack usage, but this is only observed when performing an FFT over very large polynomials.

[11] https://github.com/sachaservan/FOLEAGE-PCG.

if we are packing 3^n coefficients over $\mathbb{F}_4$, we want the output of the DPF to be close to a power of 3. To achieve this, we terminate 5 levels early and pack 512 elements of $\mathbb{F}_4$ in the virtual leaves by having the DPF output be a 1024 bit block. Therefore, the key size of each DPF is $3 \cdot 128 \cdot (n-5) + 128 + 2 \cdot 512$ when using AES with 128-bit keys. We report the SPFSS benchmarks in Table 2 when evaluating the sum of 730 DPFs (this corresponds to the $t = 27$ regime in Fig. 1, since the SPFSS needs to be instantiated with $t^2 = 729$ DPFs). When evaluating the SPFSS, we observe a roughly $1.8\times$ reduction in computation time over evaluating just one DPF. This is due to better cache performance when evaluating many DPFs and working over the same memory allocation to evaluate consecutive DPFs. Our choice of DPF range 3^{11}, 3^{13}, and 3^{15} correspond to the size of a regular noise block when $D = 3^{14}$, $D = 3^{16}$, and $D = 3^{18}$, respectively (see Table 4).

Table 2. Performance of our SPFSS (for the sum of 730 DPFs) on two EC2 instances and comparison to the raw AES computation time required for the PRG evaluations.

Range (elements of $\mathbb{F}_4$)	SPFSS.Gen (c5.metal | t2.large)	SPFSS.FullEval (c5.metal | t2.large)	AES (c5.metal | t2.large)	**Key Size** (per party)
3^{11}	5 ms | 11 ms	26 ms | 39 ms	18 ms | 27 ms	315 kB
3^{13}	7 ms | 13 ms	260 ms | 364 ms	174 ms | 253 ms	385 kB
3^{15}	8 ms | 16 ms	2357 ms | 3272 ms	1526 ms | 2229 ms	456 kB

Table 3. Performance of our FFT implementation over $\mathbb{F}_4$ on two different EC2 instances. Packing increases throughput almost linearly with the packing size. However, with a large number of variables (> 16), it is more efficient to use smaller packing values to avoid the increased memory usage from the recursive FFT function calls.

Number of Variables	Packed FFT ($4\times$) (c5.metal | t2.large)	Packed FFT ($16\times$) (c5.metal | t2.large)	Packed FFT ($32\times$) (c5.metal | t2.large)
14	20 ms | 30 ms	21 ms | 33 ms	28 ms | 45 ms
16	180 ms | 280 ms	213 ms | 329 ms	312 ms | 475 ms
18	1682 ms | 2608 ms	2165 ms | 3280 ms	4913 ms | 7478 ms

Benchmarking our PCG. Next, we benchmark the performance of the PCG from Fig. 1 on various parameters. The parameter $D = 3^n$ determines the number of Beaver triples we generate in total. In contrast, the parameters c (compression factor) and t (noise weight) influence the size of the PCG key and evaluation time. Specifically, evaluating the PCG requires $(c \cdot t)^2$ calls to the DPF on domain size D/t (due to regular noise) and $c(c+1)/2$ calls to the FFT (which we can parallelize by a factor of up to 32 using packing on 64-bit architectures). The

DPF evaluation cost ends up being the dominant factor (approximately 70%) in the total computation. The FFT accounts for less than 5% of the total computation. Interestingly, *packing* the FFT (which requires computing a matrix transpose of dimension $c(c+1)/2 \times 3^n$ to translate from $c(c+1)/2$ polynomials to a packed representation suitable for computing the FFT in parallel) accounts for 15% of the total computation! This motivates using small values of c, such as $c = 4$, as otherwise this transpose becomes the dominant cost in the entire PCG expansion. We leave exploring the possibility of implementing fast SIMD-based matrix-transpose algorithms (e.g., [4,44]) as a promising direction for future work, since it may allow using a smaller noise weight (e.g., $t = 9$) and larger c.

We set $t = 27$ since we need it to be a power of 3 (see Remark 3), and report the computational costs of the PCG for different values of D in Table 4 and c. The choice of $(c = 4, t = 27)$ corresponds to a conservative parameter choice based on our calculations, which we detail in the full version of this paper [7]. To show the influence of c on the performance, we also evaluate our PCG construction on $c = 3$, which corresponds to a more aggressive parameter choice. We observe a much smaller PCG seeds and better concrete performance with $c = 3$ compared to $c = 4$.

Table 4. Performance of our PCG implementation on two different EC2 instances. We set the noise parameter to $t = 27$ and let $c = 4$ in the left table (our conservative parameter choice) and $c = 3$ in the right table (our aggressive parameter choice); $D = 3^{18}$ ran out of memory on the `t2.large`.

(a) Parameters: $(c = 4,\ t = 27)$

D	PCG.Expand (c5.metal | t2.large)	**Key Size** (per party)
3^{14}	579 ms | 890 ms	5.0 MB
3^{16}	5.9 s | 8.4 s	6.2 MB
3^{18}	54.3 s | –	7.3 MB

(b) Parameters: $(c = 3,\ t = 27)$

D	PCG.Expand (c5.metal | t2.large)	**Key Size** (per party)
3^{14}	346 ms | 534 ms	2.8 MB
3^{16}	3.5 s | 5.2 s	3.5 MB
3^{18}	32.1 s | –	4.1 MB

Estimating Setup Costs. We use the libOTe library [42] to benchmark the state-of-the-art OT protocols. We run libOTe on `localhost` and evaluated both SoftSpoken OT [43] and the RRT' silent OT [41]. For SoftSpoken, we measured roughly 50,000,000 OT/s on the `c5.metal` machine and roughly 32,000,000 OT/s on the `t2.large`. For the RRT, we measure a throughput of nearly 7,000,000 on `c5.metal` and 4,000,000 on the `t2.large`. To run our distributed DPF key generation protocol, we require $n = 14$ (at $D = 3^{14}$) and $n = 18$ (at $D = 3^{18}$) rounds per DPF. All the $(ct)^2$ DPF keys can be computed in parallel. Therefore, in total, using our conservative parameters of $c = 4$ and $t = 27$, we require roughly 11,600 parallel calls to an OT functionality in n rounds. Our aggressive parameters of $c = 3$ and $t = 27$ only require 6,561 parallel OT calls.

Acknowledgements. We thank the anonymous reviewers for helpful suggestions. We thank Peter Rindal for help with running the libOTe [42] library, and Marcel Keller for answering our questions about Overdrive [36]. We thank Elette Boyle and Matan Hamilis for several comments that helped us improve the presentation of the paper. Geoffroy Couteau, Clément Ducros, and Dung Bui were supported by the French *Agence Nationale de la Recherche* (ANR), under grant ANR-20-CE39-0001 (project SCENE), by the France 2030 ANR Project ANR22-PECY-003 SecureCompute, and by the ERC OBELiSC. Dung Bui was supported by DIM Math Innovation 2021 (N°IRIS: 21003816) from the Paris Mathematical Sciences Foundation (FSMP) funded by the Paris Île-de-France Region. Maxime Bombar was supported by the NWO Gravitation Project QSC. Alain Couvreur was supported by the French *Agence Nationale de la Recherche* (ANR) under grant ANR-21-CE39-0009-BARRACUDA, by the *plan France 2030* under the project ANR-22-PETQ-0008 and by Horizon-Europe MSCA-DN project *Encode*.

References

1. Aguilar, C., Blazy, O., Deneuville, J.C., Gaborit, P., Zémor, G.: Efficient encryption from random quasi-cyclic codes. Cryptology ePrint Archive, Report 2016/1194 (2016), https://eprint.iacr.org/2016/1194
2. Aguilar Melchor, C., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Bos, J., Deneuville, J.C., Dion, A., Gaborit, P., Lacan, J., Persichetti, E., Robert, J.M., Véron, P., Zémor, G., Bos, J.: HQC. Round 4 Submission to the NIST Post-Quantum Cryptography Call (Oct 2022), https://pqc-hqc.org/
3. Aguilar-Melchor, C., Blazy, O., Deneuville, J.C., Gaborit, P., Zémor, G.: Efficient encryption from random quasi-cyclic codes. IEEE Transactions on Information Theory **64**(5), 3927–3943 (2018)
4. Amiri, H., Shahbahrami, A.: SIMD programming using Intel vector extensions. J. Parallel Distrib. Comput. **135**(C), 83-100 (Jan 2020). https://doi.org/10.1016/j.jpdc.2019.09.012
5. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO'91. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (Aug 1992). https://doi.org/10.1007/3-540-46766-1_34
6. Beaver, D.: Correlated pseudorandomness and the complexity of private computations. In: 28th ACM STOC. pp. 479–488. ACM Press (May 1996). https://doi.org/10.1145/237814.237996
7. Bombar, M., Bui, D., Couteau, G., Couvreur, A., Ducros, C., Servan-Schreiber, S.: FOLEAGE: $\mathbb{F}_4$OLE-based multi-party computation for boolean circuits. Cryptology ePrint Archive, Paper 2024/429 (2024), https://eprint.iacr.org/2024/429
8. Bombar, M., Couteau, G., Couvreur, A., Ducros, C.: Correlated pseudorandomness from the hardness of quasi-abelian decoding. In: CRYPTO 2023, Part IV. pp. 567–601. LNCS, Springer, Heidelberg (Aug 2023). https://doi.org/10.1007/978-3-031-38551-3_18
9. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 896–912. ACM Press (Oct 2018).https://doi.org/10.1145/3243734.3243868
10. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Resch, N., Scholl, P.: Correlated pseudorandomness from expand-accumulate codes. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part II. LNCS, vol. 13508, pp. 603–633. Springer, Heidelberg (Aug 2022). https://doi.org/10.1007/978-3-031-15979-4_21

11. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Rindal, P., Scholl, P.: Efficient two-round OT extension and silent non-interactive secure computation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 291–308. ACM Press (Nov 2019). https://doi.org/10.1145/3319535.3354255
12. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 489–518. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26954-8_16
13. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Correlated pseudorandom functions from variable-density LPN. In: 61st FOCS. pp. 1069–1080. IEEE Computer Society Press (Nov 2020). https://doi.org/10.1109/FOCS46700.2020.00103
14. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Correlated pseudorandom functions from variable-density LPN. In: 61st FOCS. pp. 1069–1080. IEEE Computer Society Press (Nov 2020). https://doi.org/10.1109/FOCS46700.2020.00103
15. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 337–367. Springer (2015)
16. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing: Improvements and extensions. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 1292–1303. ACM Press (Oct 2016). https://doi.org/10.1145/2976749.2978429
17. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. vol. 6. Association for Computing Machinery, New York, NY, USA (jul 2014). https://doi.org/10.1145/2633600, https://doi.org/10.1145/2633600
18. Cascudo, I., Cramer, R., Xing, C., Yuan, C.: Amortized complexity of information-theoretically secure MPC revisited. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part III. LNCS, vol. 10993, pp. 395–426. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96878-0_14
19. Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Math. Comput. **19**, 297–301 (1965). https://doi.org/10.1090/S0025-5718-1965-0178586-1
20. Couteau, G., Ducros, C.: Pseudorandom correlation functions from variable-density LPN, revisited. In: Boldyreva, A., Kolesnikov, V. (eds.) PKC 2023, Part II. LNCS, vol. 13941, pp. 221–250. Springer, Heidelberg (May 2023). https://doi.org/10.1007/978-3-031-31371-4_8
21. Couteau, G., Rindal, P., Raghuraman, S.: Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part III. LNCS, vol. 12827, pp. 502–534. Springer, Heidelberg, Virtual Event (Aug 2021). https://doi.org/10.1007/978-3-030-84252-9_17
22. Damgård, I., Nielsen, J.B., Nielsen, M., Ranellucci, S.: The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 167–187. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63688-7_6
23. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (Aug 2012). https://doi.org/10.1007/978-3-642-32009-5_38

24. Doerner, J., shelat, a.: Scaling ORAM for secure computation. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 523–535. ACM Press (Oct / Nov 2017). https://doi.org/10.1145/3133956.3133967
25. Even, S., Goldreich, O., Lempel, A.: A randomized protocol for signing contracts. In: Chaum, D., Rivest, R.L., Sherman, A.T. (eds.) CRYPTO'82. pp. 205–210. Plenum Press, New York, USA (1982)
26. Gilboa, N.: Two party RSA key generation. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 116–129. Springer, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48405-1_8
27. Gilboa, N., Ishai, Y.: Distributed point functions and their applications. In: Advances in Cryptology–EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings 33. pp. 640–658. Springer (2014)
28. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions, p. 241-264. Association for Computing Machinery, New York, NY, USA (2019), https://doi.org/10.1145/3335741.3335752
29. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th ACM STOC. pp. 218–229. ACM Press (May 1987). https://doi.org/10.1145/28395.28420
30. Guo, X., Yang, K., Wang, X., Zhang, W., Xie, X., Zhang, J., Liu, Z.: Half-tree: Halving the cost of tree expansion in COT and DPF. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 330–362. Springer (2023)
31. Hazay, C., Orsini, E., Scholl, P., Soria-Vazquez, E.: Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT). In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018, Part III. LNCS, vol. 11274, pp. 86–117. Springer, Heidelberg (Dec 2018). https://doi.org/10.1007/978-3-030-03332-3_4
32. Huffman, W.C., Kim, J.L., Solé, P.: Concise encyclopedia of coding theory. Chapman and Hall/CRC (2021)
33. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (Aug 2003). https://doi.org/10.1007/978-3-540-45146-4_9
34. Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. In: Proceedings of the 2020 ACM SIGSAC conference on computer and communications security. pp. 1575–1590 (2020)
35. Keller, M., Orsini, E., Scholl, P.: MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 830–842. ACM Press (Oct 2016). https://doi.org/10.1145/2976749.2978357
36. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 158–189. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78372-7_6
37. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free xor gates and applications. In: Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35. pp. 486–498. Springer (2008)
38. Oberst, U.: The fast fourier transform. SIAM journal on control and optimization **46**(2), 496–540 (2007)

39. OpenSSL Project: OpenSSL cryptography and SSL/TLS toolkit. https://www.openssl.org/, accessed: 2024-02-12
40. Rabin, M.: How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard University, (1981)
41. Raghuraman, S., Rindal, P., Tanguy, T.: Expand-convolute codes for pseudorandom correlation generators from LPN. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 14084, pp. 602–632. Springer (2023). https://doi.org/10.1007/978-3-031-38551-3_19, https://doi.org/10.1007/978-3-031-38551-3_19
42. Rindal, P., Roy, L.: libOTe: an efficient, portable, and easy to use oblivious transfer library. https://github.com/osu-crypto/libOTe
43. Roy, L.: SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part I. LNCS, vol. 13507, pp. 657–687. Springer, Heidelberg (Aug 2022). https://doi.org/10.1007/978-3-031-15802-5_23
44. Twogood, R.E., Ekstrom, M.P.: An extension of Eklundh's matrix transposition algorithm and its application in digital image processing. IEEE Trans. Comput. **25**(9), 950-952 (sep 1976). https://doi.org/10.1109/TC.1976.1674721, https://doi.org/10.1109/TC.1976.1674721
45. Yao, A.C.C.: How to generate and exchange secrets (extended abstract). In: 27th FOCS. pp. 162–167. IEEE Computer Society Press (Oct 1986). https://doi.org/10.1109/SFCS.1986.25

Perfectly-Secure Multiparty Computation with Linear Communication Complexity over Any Modulus

Daniel Escudero[1](✉), Yifan Song[2,3], and Wenhao Wang[4]

[1] J.P. Morgan AI Research & J.P. Morgan AlgoCRYPT CoE, New York, NY, USA
daniel.escudero@protonmail.com

[2] Institute for Theoretical Computer Science, Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, People's Republic of China

[3] Shanghai Qi Zhi Institute, Shanghai, People's Republic of China

[4] Yale University, New Haven, CT, USA

Abstract. Consider the task of secure multiparty computation (MPC) among n parties with perfect security and guaranteed output delivery, supporting $t < n/3$ active corruptions. Suppose the arithmetic circuit C to be computed is defined over a finite ring $\mathbb{Z}/q\mathbb{Z}$, for an arbitrary $q \in \mathbb{Z}$. It is known that this type of MPC over such ring is possible, with communication that scales as $O(n|C|)$, *assuming* that q scales as $\Omega(n)$. However, for *constant-size* rings $\mathbb{Z}/q\mathbb{Z}$ where $q = O(1)$, the communication is actually $O(n \log n|C|)$ due to the need of the so-called ring extensions. In most natural settings, the number of parties is variable but the "datatypes" used for the computation are fixed (*e.g.* 64-bit integers). In this regime, no protocol with linear communication exists.

In this work we provide an MPC protocol in this setting: perfect security, G.O.D. and $t < n/3$ active corruptions, that enjoys linear communication $O(n|C|)$, even for *constant-size* rings $\mathbb{Z}/q\mathbb{Z}$. This includes as important particular cases small fields such as $\mathbb{F}_2$, and also the ring $\mathbb{Z}/2^k\mathbb{Z}$. The main difficulty in achieving this result is that widely used techniques such as linear secret-sharing cannot work over constant-size rings, and instead, one must make use of ring extensions that add $\Omega(\log n)$ overhead, while *packing* $\Omega(\log n)$ ring elements in each extension element in order to amortize this cost. We make use of reverse multiplication-friendly embeddings (RMFEs) for this packing, and adapt recent techniques in *network routing* (Goyal *et al.* CRYPTO'22) to ensure this can be efficiently used for non-SIMD circuits. Unfortunately, doing this naively results in a restriction on the minimum width of the circuit, which leads to an extra additive term in communication of $\mathsf{poly}(n) \cdot \mathsf{depth}(C)$. One of our biggest technical contributions lies in designing novel techniques to overcome this limitation by packing elements that are distributed across *different* layers. To the best of our knowledge, all works that have a notion of packing (*e.g.* RMFE or packed secret-sharing) group gates across the same layer, and not doing so, as in our work, leads to a unique set of challenges and complications.

K.-M. Chung and Y. Sasaki (Eds.): ASIACRYPT 2024, LNCS 15489, pp. 102–132, 2025.
https://doi.org/10.1007/978-981-96-0938-3_4

1 Introduction

In secure multiparty computation, or MPC for short, a set of parties $P_1, \dots, P_n$, each having an input x_i, want to compute a function of their inputs $y = f(x_1, \dots, x_n)$ in such a way that only the output y is produced, and nothing additional about the parties' inputs is revealed. This can be done assuming a trusted party that receives the inputs, computes the function and returns the output while promising to leak nothing else, but the goal in MPC is to achieve the same guarantees without relying on such trusted party, using only communication among the parties. MPC protocols are resilient against a potential coalition of *corrupted* parties that cooperate, coordinated by an *adversary*, in order to break the remaining parties' privacy. Several MPC protocols have been proposed through time, leading to practical constructions seen in recent years and even some real-world applications and deployments (*e.g.* see https://mpc.cs.berkeley.edu/).

MPC protocols are typically characterized by the amount of corrupted parties t they can tolerate, with three notable settings being $t < n/3$, $t < n/2$ and $t < n$. Protocols with $t \geq n/3$ must allow for some negligible statistical error (if $t < n/2$), or even make use of cryptographic assumptions (if $t \geq n/2$). In the $t < n/3$ case several results are known: not only we can obtain perfectly secure MPC [BOGW88], which safeguards the parties' inputs regardless of the computational power of the adversary, but we can do so with several appealing features such as guaranteed output delivery (G.O.D., which ensures the successful completion of the protocol in spite of any adversarial misbehavior), and a communication complexity that requires each party to send in average an amount of messages that is independent of the total number of parties, and depends only on the circuit size of the function f (here we omit additive terms that are independent of the circuit size but only depend on the number of parties) [GLS19]. More precisely, we can represent the function f as an arithmetic circuit C comprised of input, output, addition and multiplication gates over some finite ring $\mathbb{Z}/q\mathbb{Z}$ of integers modulo an integer q, and if $|C|$ denotes the number of multiplication gates, it is possible to obtain perfectly secure MPC over $\mathbb{Z}/q\mathbb{Z}$ with G.O.D. where the total communication complexity is $O(n|C|)$ ring elements, which we refer to as *linear* communication complexity. A common choice is taking q to be a prime, but this claim holds for any *arbitrary* integer q: the *Chinese remainder theorem* reduces the general case to moduli of the form $q = p^k$, for a prime p and an integer k, and works such as [ACD+19] show that perfectly secure MPC with G.O.D. over these moduli is possible, with linear communication complexity. It is relevant to study the general modulus case as it contains several relevant cases such as $q = 2$, 2^{32} or 2^{64}, which have received considerable attention in recent works due to practical benefits [DEF+19].

Unfortunately, the claim of linear communication is not entirely accurate. Shamir secret-sharing is a core technique to enable these results, and for this scheme to work over $\mathbb{Z}/p^k\mathbb{Z}$, p must be strictly larger than n. Once n becomes larger than p, a so-called *Galois ring extension* of degree $\Omega(\log n)$ must be used. It is still true that the communication is $O(n|C|)$ ring elements, but this time each ring element is $\Omega(\log n)$ bits long. Hence, asymptotic communication is truly $O(n \log n|C|)$. In [CCXY18], the authors partially address this issue relying on the technique of reverse multiplication-friendly embeddings (RMFEs), which allow them to remove the $\log n$ overhead when computing many copies of the same circuits, i.e., a SIMD circuit. However, for general circuits, directly applying their techniques does not work as we will discuss in Sect. 1.3. This leads to the following question:

Can we design perfectly secure MPC protocols for general circuits over a constant-size ring $\mathbb{Z}/q\mathbb{Z}$ whose asymptotic communication complexity scales as $O(n|C|)$?

Again, all current solutions have communication that scales as $O(n|C|)$ ring elements, but assuming that either the ring bit-size scales as $\log n$ or the underlying circuit is a SIMD circuit. For fixed-size rings (which is the natural setting in practice) and general circuits, ring extensions must be used, which leads to a communication of $O(n \log n|C|)$. Recall that this affects the very practically-motivated settings of binary computation (*i.e.* circuits over $\mathbb{Z}_2$), and also the `int32` and `int64` cases (*i.e.* circuits over $\mathbb{Z}/2^k\mathbb{Z}$ for $k = 32$ and $k = 64$).

1.1 Our Contribution

In this work we give an answer in the affirmative to the aforementioned question. We provide an MPC protocol with the following desirable features:

- Perfect security against an active adversary corrupting $t < n/3$ parties
- Securely computes circuits over $\mathbb{Z}/q\mathbb{Z}$ for *any constant* q
- The total number of elements in $\mathbb{Z}/q\mathbb{Z}$ communicated scales as $O(|C| \cdot n + c \cdot n \cdot \log n + n^3 \cdot \log^2 n)$, where c is the number of clients
- Guaranteed output delivery

As we have previously discussed, via CRT this task reduces to the task of computation over a ring $\mathbb{Z}/p^k\mathbb{Z}$ with constant p. For the sake of presentation we focus on $p = 2$, that is, computation over $\mathbb{Z}/2^k\mathbb{Z}$, but the ideas presented directly work for more general p.

Remark 1 (On communication complexity). In our protocol, the communication complexity scales as $O(|C| \cdot n)$ only when $|C| = \Omega(c \log n + n^2 \log^2 n)$.

Remark 2 (On security with abort). We note that even removing the G.O.D. condition, and settling with security with abort only, obtaining a protocol with the other properties is not simple, and is on its own a relevant and challenging open question. In this work we aim for the stronger notion of G.O.D.

Remark 3 (Cost of addition gates). In our work, (a linear amount of) communication is required for every *addition gate*, which is not the case if one settles for $O(n \log n|C|)$ complexity (where addition gates are for free in terms of communication). Looking ahead, this occurs due to a technique called *network routing*, originated in [GPS22, GPS21], which is used to exploit some notion of *packing* while routing values through the circuit correctly. The work of [GPS22] (which is set in a different context than ours) also suffers from communication per addition gates, and avoiding this overhead in our work would likely lead to improvements to [GPS22]. If the circuit does not have substantially many more addition gates than multiplication gates (a reasonable setting in practice), then this condition becomes immaterial for our asymptotic $O(n|C|)$ claim.

1.2 Related Work

Perfectly secure MPC for $t < n/3$ with linear communication and G.O.D. has been studied in multiple works such as the one by Goyal, Liu, and Song [GLS19] and Beerliová-Trubíniová and Hirt [BTH08]. These are set specifically in the context of finite fields $\mathbb{F}_{p^d}$, where $p^d = \Omega(n)$. The work of Abspoel et al. [ACD+19] generalizes this to the ring $\mathbb{Z}/p^k\mathbb{Z}$ by using Galois ring extensions of degree $d = \Omega(n)$. Unfortunately, as we have pointed out, these techniques are not suitable for *constant-sized* rings since they add an $\Omega(\log n)$ overhead.

If it is known that t is *far* from $n/3$, that is, $t < n(\frac{1}{3}-\epsilon)$ for any constant $\epsilon > 0$, then it is possible to obtain perfectly fully secure MPC with a communication of $O(|C|)$ (independent of n) for fields $\mathbb{F}_{p^d}$ with $p^d = \Omega(n)$, and $O(\log n|C|)$ for constant-sized fields [DIK10]. These techniques extend naturally to the case of $\mathbb{Z}/p^k\mathbb{Z}$ by using ring extensions as in [ACD+19], and to arbitrary $\mathbb{Z}/q\mathbb{Z}$ via CRT. This is better than our linear communication $O(n|C|)$, but it assumes the aforementioned gap $\epsilon > 0$. Our protocol works for the case $\epsilon = 0$: $t < n/3$, and n can be as small as $n = 3t + 1$.

In the $t < n/2$ case with *statistical* security, a similar situation exists: most protocols with G.O.D. that achieve linear communication complexity require a ring $\mathbb{Z}/p^k\mathbb{Z}$ with $p > n$ [BFO12, GSZ20, ACD+19]. This state of affairs changed for the case of $\mathbb{F}_p$ and *security with abort* for *constant* p in the recent work of [PS21]. The same ideas in [PS21] extend naturally to $\mathbb{Z}/p^k\mathbb{Z}$ for constant p. However, it is not clear how to extend the ideas in [PS21] to G.O.D. since, in the $t < n/2$ regime, the techniques used to achieve full security are considerably much more complex than these for $t < n/3$ (which are already quite intricate). The core difference is that in $t < n/2$ setting the central idea to achieve G.O.D. , dispute control, differs from player elimination—in spite of sharing some similarities: unlike player elimination, in $t < n/2$ this pair cannot be removed because one may be inadvertently removing one of the $t + 1$ honest parties, and t honest parties alone cannot have enough joint information to finish the computation.

Finally, we note that if one only wants to achieve statistical security (rather than perfect security) in the G.O.D. setting for $t < n/3$, [IKP+16] has achieved $O(\text{poly}(\log(n))|C|)$ elements of communication if $t < n(1/2-\epsilon)$, for any constant $\epsilon > 0$, which is even better than linear communication.

1.3 Overview of Our Techniques

We begin by highlighting the difficulties that existing works face when considering MPC over constant-size rings. Perfectly secure MPC with linear communication (for non-constant fields) was first proposed in [BTH08], and it was later improved in [GLS19] to remove a term dependent on the circuit depth. These ideas can be generalized to rings such as $\mathbb{Z}/p^k\mathbb{Z}$ [ACD+19], again with the same complexity if $p = \Omega(n)$. This complexity comes from the use of Shamir secret-sharing, which requires enough interpolation points to operate, and hence it requires a ring of large enough characteristic. Shamir's is not the only linear secret-sharing scheme one could use, but it is unlikely there exist other schemes that somehow reduce this secret/share size requirement, since this is highly connected to the MDS conjecture (see for example [FR22, Lemma 1]). Since Shamir secret-sharing seems unavoidable, our core idea is to use the ring extensions needed for it, while *packing* multiple entries in a single ring extension secret-shared value, so that the cost per single share is ultimately linear. Ultimately, the challenges lie on efficiently making use of this packing, which is our focus in this overview.

As we have mentioned previously, here and for the rest of our paper we focus on the case $\mathbb{Z}/2^k\mathbb{Z}$. Furthermore, for the sake of this introduction only we focus on security with abort (which, as mentioned previously, is already a challenging task on its own). We highlight towards the end of the section how to tackle the G.O.D. case. Our work makes use of several techniques in the literature such as player elimination [BTH08], reverse multiplication-friendly embeddings (RMFEs) [CCXY18], network routing [GPS22, GPS21], among others. In this section we provide a high level overview of how these ideas are put together in order to obtain our protocol. First, we introduce some notation, with further details given in Sect. 2 and beyond. We let $\mathsf{GR}(2^k, m)$ be a Galois ring extension of $\mathbb{Z}/2^k\mathbb{Z}$ of degree m, which can be seen as the ring of polynomials over $\mathbb{Z}/2^k\mathbb{Z}$ modulo a monic polynomial irreducible mod 2 of degree m (for more details see Sect. 2 or [Wan03]). We can perform Shamir secret-sharing over this ring if $m = \lceil \log n \rceil + 1$ [ACD+19], and we denote degree-d sharings of $x \in \mathsf{GR}(2^k, m)$ by $[x]_d$. An RMFE is a pair of $\mathbb{Z}/2^k\mathbb{Z}$-linear homomorphisms $(\phi : (\mathbb{Z}/2^k\mathbb{Z})^\ell \to \mathsf{GR}(2^k, m),\ \psi : \mathsf{GR}(2^k, m) \to (\mathbb{Z}/2^k\mathbb{Z})^\ell)$ satisfying $\psi(\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{y})) = \boldsymbol{x} \star \boldsymbol{y}$ for every $\boldsymbol{x}, \boldsymbol{y} \in (\mathbb{Z}/2^k\mathbb{Z})^\ell$. Such objects exist, with $\ell = \Theta(m)$ [CCXY18, ELXY23], and in fact we can also take them such that $\phi(\mathbf{1}) = 1$ [ELXY23]. This ensures that $\psi(\phi(\boldsymbol{x})) = \psi(\phi(\boldsymbol{x})\phi(\mathbf{1})) = \boldsymbol{x} \star \mathbf{1} = \boldsymbol{x}$.

Embedding Using RMFEs. We may take as a starting point the approach in [CCXY18] in order to secret-share elements of $\mathbb{Z}/2^k\mathbb{Z}$ efficiently using Shamir secret-sharing via RMFEs. Recall that one (naive) way of secret-sharing an element $x \in \mathbb{Z}/2^k\mathbb{Z}$ using Shamir SS is to embed it in $\mathsf{GR}(2^k, m)$, and then secret-sharing over $\mathsf{GR}(2^k, m)$, which results in an undesired overhead of $m \approx \log n$. Instead, one may secret-share m elements simultaneously by interpreting them as an element of $\mathsf{GR}(2^k, m)$, removing the overhead of m in an amortized sense (*i.e.* after dividing by the number of secrets m), but unfortunately this approach does not interact well with the MPC setting since one cannot easily multiply elements this way.

RMFEs are introduced in [CCXY18] as a solution to this problem. Instead of secret-sharing m elements by thinking of them as an element of $\mathsf{GR}(2^k, m)$, ℓ elements $\boldsymbol{x} = (x_1, \ldots, x_\ell) \in (\mathbb{Z}/2^k\mathbb{Z})^\ell$ are shared together by first mapping them as $\phi(\boldsymbol{x}) \in \mathsf{GR}(2^k, m)$, and sharing $[\phi(\boldsymbol{x})]_t$ instead. This has an overhead of m for ℓ secrets, which is $m/\ell = \Theta(1)$ amortized. Furthermore, the multiplicative property of RMFEs turns out to enable products on secret-shared data, making this embedding particularly suitable for MPC. In fact, this is used in [CCXY18] to remove the $\log n$ overhead of perfect security in the context in which the circuit C is structured as ℓ copies of the same function, run on possibly different inputs. Our goal however is to enable a more general class of circuits that have no specific structure in terms of their wiring, and hence this approach is insufficient.

To illustrate the main challenge, let us discuss how the approach from [CCXY18] works, for ℓ copies of the same circuit. The invariant the parties maintain is that, for every set of ℓ values $\boldsymbol{x} = (x_1, \ldots, x_\ell) \in (\mathbb{Z}/2^k\mathbb{Z})^\ell$ corresponding to the same wire across the ℓ copies, the parties have sharings $[\phi(\boldsymbol{x})]_t$. Given ℓ copies of a multiplication gate with inputs $\boldsymbol{x}$ and $\boldsymbol{y}$, the invariant is maintained by using a triple $([\phi(\boldsymbol{a})]_t, [\phi(\boldsymbol{b})]_t, [\phi(\boldsymbol{c})]_t)$, where $\boldsymbol{c} = \boldsymbol{a} \star \boldsymbol{b}$. Assuming the parties have $[\phi(\boldsymbol{x})]_t$ and $[\phi(\boldsymbol{y})]_t$, they can locally compute $[\phi(\boldsymbol{x}) + \phi(\boldsymbol{a})]_t$ and open this as $\phi(\boldsymbol{u})$, and similarly open $\phi(\boldsymbol{v})$ where $\boldsymbol{v} = \boldsymbol{y} + \boldsymbol{b}$. Crucially, since this is Shamir reconstruction, this can be done with error correction/detection, ensuring no errors are introduced. Now, the parties compute locally

$$[z]_t = \phi(\boldsymbol{u}) \cdot \phi(\boldsymbol{v}) - \phi(\boldsymbol{v}) \cdot [\phi(\boldsymbol{a})]_t - \phi(\boldsymbol{u}) \cdot [\phi(\boldsymbol{b})]_t + [\phi(\boldsymbol{c})]_t,$$

which satisfies $\psi(z) = \boldsymbol{x} \star \boldsymbol{y}$, thanks to the properties of RMFEs. Finally, the parties can execute a simple re-encoding protocol that applies $\phi \circ \psi : \mathsf{GR}(2^k, m) \to \mathsf{GR}(2^k, m)$ to $[z]_t$, yielding $[\phi(\boldsymbol{x} \star \boldsymbol{y})]_t$, hence preserving the desired invariant.

Now, if the circuit is non-SIMD, it could easily happen that, say, a value encoded in position 1 must be multiplied (or even added) with a value encoded in position 2. The properties of RMFEs only allow for the computation of $\boldsymbol{x} \star \boldsymbol{y} = (x_1 y_1, \ldots, x_\ell y_\ell)$, but they fall short if one somehow needs products that are "not aligned", such as $(x_1 y_2, x_2 y_1, x_3 y_3, \ldots)$. SIMD circuits do not present such misalignments, which is why the techniques in [CCXY18] work in that setting.

Network Routing. To alleviate this issue, we may resort to network routing, a general technique introduced in [GPS22, GPS21] to ensure the sharings of output groups can be rearranged in such a way that they are "aligned" when fed as inputs to future groups of gates. In these works, such "wiring" issues appeared in the context of packed secret-sharing where, as in our setting, parties have shares whose underlying secrets are *vectors* that can somehow be added or multiplied component-wise, but cannot be re-routed easily. In our work we show that such techniques can also be used in our context, where the "packing" is done with RMFEs. For the sake of keeping this overview as lightweight as possible, we will not dive into the details on how this is done, and instead we will refer the reader to Sects. 3.4 and ?? where details are provided. For now, it suffices to say that there is an efficient method for the parties to obtain sharings $[\phi(\boldsymbol{x})]_t$ for every input group in a given layer, starting from packed sharings of every output group of the previous layers.

For simplicity in the exposition we will assume from now on that the circuit only has multiplication gates, without any addition operation. We discuss at the end of this overview why this is convenient, and how to handle the case of addition gates as well. Naturally, our main protocols include the general case.

On the Dependency on Circuit Width. Using the network routing techniques mentioned above, together with the protocol sketched earlier, we would obtain the desired result: perfectly secure MPC for $t < n/3$ over $\mathbb{Z}/2^k\mathbb{Z}$ with G.O.D. and linear communication $O(n)$. However, this result hides an assumption on the circuit structure: it requires each layer to contain at least $\Omega(n\ell)$ multiplication gates. This is because every layer requires parallel reconstructions for each of its multiplication gates, and robust reconstruction of Shamir sharings with *linear* communication (*i.e.* instead of all parties sending their shares to each other, which would be quadratic) requires $\Omega(n)$ reconstructions to be done in parallel [DN07]. On top of this each such Shamir sharing "packs" $\ell \approx O(\log n)$ values, which results in a requirement of $n\ell$ secrets to be reconstructed (per layer!) to obtain the communication gains.

The above results in a total communication that is not $O(n|C|)$, but rather $O(n|C| + n^2\ell \cdot \mathsf{depth}(C))$. For circuits C such that $\mathsf{depth}(C) \gg |C|/n$ (*e.g.* "skinny" circuits), this extra term dominates communication. In contrast, for the case of MPC over $\mathbb{F}_p$ for large p, it is known that the $n^2\ell \cdot \mathsf{depth}(C)$ term can be eliminated, resulting in true linear communication $O(n|C|)$ [GLS19]. This leaves a gap between what we know over (large) fields (which can be generalized for $\mathbb{Z}/p^k\mathbb{Z}$ for large p), and the case of $\mathbb{Z}/2^k\mathbb{Z}$. What follows is dedicated to discuss how we address this complexity gap.

Computing the Circuit Optimistically with Additive Secret-Sharing. We first reduce the term $n^2\ell \cdot \mathsf{depth}(C)$ to $n\ell \cdot \mathsf{depth}(C)$ via the following core idea—also used in [GLS19]: derive *additive* sharings from Shamir sharings and use the additive sharings to compute the circuit (which enables cheating but can be done with linear communication without any minimum batch size requirement), and only use the Shamir sharings for a final verification check (which involves reconstructing many robust sharings in parallel and hence can be done with linear communication complexity). For the optimistic computation of the circuit we will make use of additive secret-sharing, which we denote by $\langle x \rangle$ for $x \in \mathbb{Z}/2^k\mathbb{Z}$ (we also extend this notation naturally to vectors over $\mathbb{Z}/2^k\mathbb{Z}$). For input gates, each client having a group of inputs $\boldsymbol{x} \in (\mathbb{Z}/2^k\mathbb{Z})^\ell$ secret-shares $[\phi(\boldsymbol{x})]_t$ towards the parties (the parties perform degree checks that ensure the degree is indeed $\leq t$), and then the parties locally derive $\langle \boldsymbol{x} \rangle$ from $[\phi(\boldsymbol{x})]_t$. This is done by first locally converting $[\phi(\boldsymbol{x})]_t$ to $\langle \phi(\boldsymbol{x}) \rangle$ (a standard procedure involving each party multiplying locally by certain Lagrange coefficients), followed by a *local* application of the mapping ψ, to obtain $\langle \psi(\phi(\boldsymbol{x})) = \boldsymbol{x} \rangle$. Now, the parties optimistically compute the circuit. For every pair of values to be multiplied $\langle x \rangle$ and $\langle y \rangle$, letting $i \in [\ell]$ be the index of this gate in its group, the parties can use the triple $([\phi(\boldsymbol{a})]_t, [\phi(\boldsymbol{b})]_t, [\phi(\boldsymbol{c})]_t)$ associated to this group to locally derive the additively shared triple $(\langle a_i \rangle, \langle b_i \rangle, \langle c_i \rangle)$, which they can use to multiply $\langle x \rangle$ and $\langle y \rangle$: open $u_i = x + a_i$ and $v_i = y + b_i$ (with linear communication without any minimum batch size restriction), and compute $\langle xy \rangle = u_i \cdot v_i - v_i \cdot \langle a \rangle - u_i \cdot \langle b \rangle + \langle c \rangle$.

For the check, recall that the parties have Shamir sharings of $[\phi(\boldsymbol{x})]_t$ for every group $\boldsymbol{x}$ in the input layer. Consider a multiplication group in the first layer having as inputs $\boldsymbol{x}, \boldsymbol{y}$. Using network routing, the parties can obtain Shamir sharings $[\phi(\boldsymbol{x})]_t, [\phi(\boldsymbol{y})]_t$, so they can compute $[\phi(\boldsymbol{u})]_t = [\phi(\boldsymbol{x})]_t + [\phi(\boldsymbol{a})]_t$ and $[\phi(\boldsymbol{v})]_t = [\phi(\boldsymbol{y})]_t + [\phi(\boldsymbol{b})]_t$, hence obtaining robust versions of the reconstructed values $\boldsymbol{u}, \boldsymbol{v}$ when these gates were computed optimistically. Furthermore, the parties can compute locally $[z]_t = \phi(\boldsymbol{u}) \cdot \phi(\boldsymbol{v}) - \phi(\boldsymbol{v}) \cdot [\phi(\boldsymbol{a})]_t - \phi(\boldsymbol{u}) \cdot [\phi(\boldsymbol{b})]_t + [\phi(\boldsymbol{c})]_t$, and apply the re-encoding protocol from [CCXY18] to obtain $[\phi(\boldsymbol{x} \star \boldsymbol{y})]_t$. Doing this for all groups in the first layer ensures that the parties obtain robust sharings $[\phi(\boldsymbol{z})]_t$ of all output groups $\boldsymbol{z}$ in the first layer. This process can be iterated: the parties use network routing to obtain robust sharings of all input groups in the second layer, obtain robust sharings of all reconstructed values $\boldsymbol{u}$ and $\boldsymbol{v}$ during the optimistic computation of this layer, and then compute robust sharings of the outputs groups in this layer. This is repeated until the output layer is reached. Before reconstructing the outputs, however, the parties robustly reconstruct in parallel all Shamir sharings $[\boldsymbol{u}]_t$ and $[\boldsymbol{v}]_t$ corresponding to *all* multiplication groups, cross checking with the values reconstructed optimistically.[1] The key point is that this robust reconstruction involves enough values as to be able to enjoy amortized linear communication complexity.

Packing Across Different Multiplication Layers. Using additive SS optimistically during the circuit computation shaves a factor of n in the term $n^2\ell \cdot \mathsf{depth}(C)$, but it still leaves us with a term $n\ell \cdot \mathsf{depth}(C)$, since we still need to RMFE-pack ℓ gates across the *same* layer. To address this we allow groups to contain gates spanning over potentially *different* layers, with the only restriction being that the set of groups must form a DAG (we say there is an edge from group A to group B if at least one gate in A connects to one gate in B; in Sect. 3.1 we show that such grouping can always be done for *any* circuit—without any width restrictions). For notational convenience we assume that the gates in every group are indexed in increasing topological order, meaning that if a gate indexed by i in a group depends on the output of another gate indexed by j in the same group, then $j < i$. As before, we still group each client's input wires into batches of size ℓ each,[2] and we also let each client secret-shares a group of inputs $\boldsymbol{x}$ as $[\phi(\boldsymbol{x})]_t$.

[1] As shown in [GLS19], certain care is needed to ensure that security is not broken by delaying verification. We omit these details in this overview.

[2] We assume each client has $\Omega(\ell) = \Omega(n)$ inputs. Otherwise there is a minor overhead due to packing, but this is only restricted to the input layer.

Careful observation reveals that such relaxation in grouping does not really affect the optimistic computation using additive SS: the parties can still derive additive SS triples $(\langle a_i \rangle, \langle b_i \rangle, \langle c_i \rangle)$ and use these to perform multiplications. However, complications will arise at the verification stage. To illustrate this, let us consider the first multiplication group in topological order, and let us denote its inputs as $\boldsymbol{x}$ and $\boldsymbol{y}$. If it was the case that all of the multiplications in the group belonged to the first layer, then the parties could perform network routing as before to obtain $[\phi(\boldsymbol{x})]_t$ and $[\phi(\boldsymbol{y})]_t$. Unfortunately, since we are packing across different layers, it can happen that, say, the left input x_i to the i-th gate is equal to the output $x_j y_j$ of the j-th gate in the same group, with $j < i$. This prevents us from determining $[\phi(\boldsymbol{x})]_t$ from the input layer alone.

Determining $[\phi(\boldsymbol{x})]_t$ *for the First Multiplication Group.* For illustration, let us keep our focus on the first multiplication group. Our goal is to show how the parties can obtain robust sharings $[\phi(\boldsymbol{x})]_t$, which, as illustrated above, cannot be derived from the input layer alone since the left input x_i to get i depends on the j-th output $x_j y_j$ of gate $j < i$. First, recall that the parties have opened $\langle \boldsymbol{u} \rangle = \langle \boldsymbol{x} \rangle + \langle \boldsymbol{a} \rangle$ and $\langle \boldsymbol{v} \rangle = \langle \boldsymbol{y} \rangle + \langle \boldsymbol{b} \rangle$, in the process of processing this group optimistically. However, due to the non-robustness of additive SS, these reconstructions may have resulted in $\boldsymbol{u}' = \boldsymbol{u} + \boldsymbol{\delta}$ and $\boldsymbol{v}' = \boldsymbol{v} + \boldsymbol{\epsilon}$, for some possibly non-zero $\boldsymbol{\delta}$ and $\boldsymbol{\epsilon}$. Using these reconstructions, the parties can compute locally $[\phi(\boldsymbol{x}+\boldsymbol{\delta})]_t = \phi(\boldsymbol{u}') - [\phi(\boldsymbol{a})]_t$ and $[\phi(\boldsymbol{y} + \boldsymbol{\epsilon})]_t = \phi(\boldsymbol{v}') - [\phi(\boldsymbol{b})]_t$, which correspond to robust sharings of the inputs $\boldsymbol{x}$ and $\boldsymbol{y}$, but *potentially incorrect.* Now we let the parties execute *any* correct multiplication protocol (we borrow the protocol from [BTH08, ACD+19] for this purpose) to compute the product $[\phi(\boldsymbol{x} + \boldsymbol{\delta}) \cdot \phi(\boldsymbol{y} + \boldsymbol{\epsilon})]_t$, followed by re-encoding as in [CCXY18] (*i.e.* applying $\phi \circ \psi$) to obtain $[\phi((\boldsymbol{x}+\boldsymbol{\delta}) \star (\boldsymbol{y}+\boldsymbol{\epsilon}))]_t$.

For simplicity let us assume that i is the only index in this group whose corresponding gate depends on other outputs from the same group, with all the other gates receiving inputs directly from the input layer. In particular, both x_j and y_j come from the input layer. Recall that the goal is to obtain $[\phi(\boldsymbol{x})]_t$ using network routing. Since the parties have robust sharings of all groups in the input layer, they have almost all the pieces needed to obtain $[\phi(\boldsymbol{x})]_t$, with the only missing part being robust sharings that contain the output of the j-th gate, since this is needed for the i-th left input x_i. Our idea is to use, for the missing j-th output, the sharings $[\phi((\boldsymbol{x} + \boldsymbol{\delta}) \star (\boldsymbol{y} + \boldsymbol{\epsilon}))]_t$ computed above, which contain the j-th output $x_j y_j + \gamma$, where $\gamma = x_j \epsilon_j + y_j \delta_j + \delta_j \epsilon_j$, in the j-th entry. In other words, the parties perform network routing on the sharings from the input layer *and* the sharing $[\phi((\boldsymbol{x} + \boldsymbol{\delta}) \star (\boldsymbol{y} + \boldsymbol{\epsilon}))]_t$ to obtain $[\phi(\boldsymbol{x})]_t$. However, since the i-th entry corresponds to $x_i + \gamma$, the actual secret is $[\phi(\boldsymbol{x} + \gamma \boldsymbol{e}_i)]_t$. In other words, the parties do not obtain robust sharings of the *correct* $\boldsymbol{x}$, but instead, the i-th entry is shifted by γ.

This may raise a red flag at first sight: recall we are using the sharing $[\phi(\boldsymbol{x} + \gamma \boldsymbol{e}_i)]_t$ to verify the multiplications in the first group, in particular, verifying that $\langle \boldsymbol{u} \rangle = \langle \boldsymbol{x} \rangle + \langle \boldsymbol{a} \rangle$ was opened correctly. However, we are using an incorrect $[\phi(\boldsymbol{x} + \gamma \boldsymbol{e}_i)]_t$, so it may be the case that this somehow helps the adversary reconstruct $\langle \boldsymbol{u} \rangle$ incorrectly. For example, the adversary may be able to reconstruct $\langle \boldsymbol{u} \rangle$ as

$\boldsymbol{u} + \gamma \boldsymbol{e}_i$, which is consistent with the robust sharings held by the parties $[\phi(\boldsymbol{x} + \gamma \boldsymbol{e}_i) + \phi(\boldsymbol{a})]_t$, and hence the check will pass, in spite of $\langle \boldsymbol{u} \rangle$ being reconstructed incorrectly. Yet, observe that this is an attack only if γ is non-zero, for which it must be the case that either δ_j or ϵ_j is not zero; say for simplicity $\delta_j \neq 0$. Fortunately, this will be caught in the check: the parties have the sharings $[\phi(\boldsymbol{x} + \gamma \boldsymbol{e}_i) + \phi(\boldsymbol{a})]_t$, which are incorrect in position i but, crucially, are correct in position j, so the adversary will not be able to conceal the fact that the j-th entry was modified.

Determining $[\phi(\boldsymbol{x})]_t$ *for Every Input Group.* The principle above applies more generally. After performing optimistic computation using additive secret-sharing, the parties perform the following for every multiplication group with inputs $\boldsymbol{x}, \boldsymbol{y}$. Let $([\phi(\boldsymbol{a})]_t, [\phi(\boldsymbol{b})]_t, [\phi(\boldsymbol{c})]_t)$ be the Shamir triple associated to the group, and recall that the parties reconstructed (potentially incorrectly) $\boldsymbol{u}$ and $\boldsymbol{v}$ as part of the optimistic multiplications. The parties compute locally $[\phi(\boldsymbol{x})]_t = \phi(\boldsymbol{u}) - [\phi(\boldsymbol{a})]_t$ and $[\phi(\boldsymbol{y})]_t = \phi(\boldsymbol{v}) - [\phi(\boldsymbol{b})]_t$, and then they compute the product $[\phi(\boldsymbol{x} \star \boldsymbol{y})]_t$ using a secure multiplication protocol followed by re-encoding, as illustrated earlier. At this point, for every set of outputs $\boldsymbol{x}$ of a given group, the parties hold $[\phi(\boldsymbol{x})]_t$. Moreover, the following crucial property holds: if no cheating occurred up to (and including) the optimistic evaluation of the gate at index i, then $\boldsymbol{x}[i]$ holds the correct wire value.

Now, the parties apply network routing to map all the packed sharings of the output groups into packed sharings of input groups $[\phi(\boldsymbol{x})]_t, [\phi(\boldsymbol{y})]_t$ for every multiplication group with inputs $\boldsymbol{x}, \boldsymbol{y}$. Importantly, due to the property above, if no cheating has occurred prior to the computation of, say, the i-th gate in a group with inputs $\boldsymbol{x}, \boldsymbol{y}$, then we know that the sharings $[\phi(\boldsymbol{x})]_t, [\phi(\boldsymbol{y})]_t$ derived from the network routing satisfy that the i-th entries x_i and y_i are *correct*. This way, if cheating occurs for the first time in this gate, by reconstructing incorrect $u'_i = x_i + a_i + \delta_i$ and $v'_i = y_i + b_i + \epsilon_i$ (which may result in more errors in other entries of the vectors $[\phi(\boldsymbol{x})]_t, [\phi(\boldsymbol{y})]_t$, but only with indices $j > i$, not for $j \leq i$), the parties can use $[\phi(\boldsymbol{x})]_t, [\phi(\boldsymbol{y})]_t$ (together with the associated triple $([\phi(\boldsymbol{a})]_t, [\phi(\boldsymbol{b})]_t, [\phi(\boldsymbol{c})]_t)$) to check the correctness of the reconstructed u'_i and v'_i.

Dealing with Addition Gates. Recall we assumed for simplicity that the circuit did not have any addition gates. We briefly comment how the general case is handled. First, both addition and multiplication gates are grouped in sets of ℓ gates each (where the gates within each group are of the same type). The optimistic computation phase remains the same: the parties handle addition gates by simply adding their (additive) shares together, locally. For the verification step, however, we need the parties to communicate for every group of addition gates. To see why this is the case, consider a group of addition gates with inputs $\langle \boldsymbol{x} \rangle, \langle \boldsymbol{y} \rangle$, and imagine that every output of these gates is later each fed to a multiplication gate. The parties can of course add locally $\langle \boldsymbol{x} \rangle, \langle \boldsymbol{y} \rangle$, but recall that for the network routing phase to work, we need the parties to have *packed Shamir* sharings of the outputs, like $[\phi(\boldsymbol{x} + \boldsymbol{y})]_t$, but it is not clear how they can obtain these from $\langle \boldsymbol{x} \rangle, \langle \boldsymbol{y} \rangle$ alone.

For the case of multiplication gates, this was achieved with the help of the (packed) triple that was used for the product. For the case of addition gates, we will use a similar idea: we make use of an *additive triple* $([\phi(\boldsymbol{a})]_t, [\phi(\boldsymbol{b})]_t, [\phi(\boldsymbol{c})]_t)$, where $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$, ask the parties to open $\boldsymbol{u} \leftarrow \langle \boldsymbol{x} + \boldsymbol{a} \rangle$ and $\boldsymbol{v} \leftarrow \langle \boldsymbol{y} + \boldsymbol{b} \rangle$, and compute $[\phi(\boldsymbol{x}+\boldsymbol{y})]_t = (\boldsymbol{u}+\boldsymbol{v}) - [\phi(\boldsymbol{c})]_t$. This can also be seen as adding an extra step that first converts $\langle \boldsymbol{x} \rangle$ to $[\phi(\boldsymbol{x})]_t$ using the "double sharing" $(\langle \boldsymbol{a} \rangle, [\phi(\boldsymbol{a})]_t)$ (and similarly for $\boldsymbol{y}$), and then add these sharings together.

On Guaranteed Output Delivery. Finally, we comment on how the protocol sketched here is extended to G.O.D., without blowing communication. We use the player elimination framework from [BTH08], in which the parties not only perform a check but, in case of failure, identify a so-called *semi-corrupted pair* in the process, which is a pair of parties that is guaranteed to contain at least one corrupted party. At this point, the pair can be safely removed from the computation (which preserves the $t < n/3$ ratio), and the computation can be restarted. Restarting the computation many times can cause communication to blow up by a large factor. To address this, the circuit is split into *segments* of certain size, and the check described here is performed at the end of each such segment, rather than at the end of the whole circuit. Setting segment sizes appropriately reduces the size of the repeated computations, which keeps communication within $O(n|C|)$. There are subtle issues, like part of the output of a group of a given segment being fed to the same segment, while some other part is fed to a future segment. This is mostly inconvenient notation-wise, but it does not add heavy technical complications.

To give a more complete picture, it remains to describe more clearly how the parties can identify a semi-corrupted pair during our check. Recall that our verification consists, in essence, of opening sharings of the form $[\phi(\boldsymbol{z})]_t$, and comparing against a previously opened set of values $\boldsymbol{z}'$. How should the parties react in case some mismatch is found? The core idea is to pinpoint to the party who announced an incorrect (additive) sharing in the first place. The main challenge with this is that, even though the parties have a robust version of the underlying *secrets* $\boldsymbol{z}$, they do not necessarily have a robust version of the additive sharings that each party should have sent when reconstructing $\boldsymbol{z}$, so it is not obvious how to identify which party sent an incorrect additive share. Fortunately, as it turns out, it is indeed possible to derive robust sharings that somehow commit the parties to the additive shares they should send at reconstruction. For this, we introduce a notion of *extended* additive sharings, which expands additive SS with the necessary information to check whether a party sent a correct additive share. We provide details in the full version of the paper.

Remark 4 (On sharings of zero). We remark that our overview here is a simplified version of our actual protocol, which must use of several other ingredients not discussed here. One of these is that, in several places, the parties need to re-randomize certain sharings using shares of zero, which is crucial for, among different purposes, preventing leakage of sensitive information when reconstructing optimistically, as in [GLS19].

1.4 Outline of the Document

We begin by presenting several important preliminaries in Sect. 2. This is followed by Sect. 3, which contains our protocols for optimistically evaluating a segment (Sect. 3.2), as well as verifying the computation is performed correctly (Sect. 3.5). This includes the network routing needed to compute robust sharings of groups in the circuit (Sect. 3.4), as well as our method to identify semi-corrupted parties once an attack has been detected (Sect. 3.6). We also discuss in detail how the circuit is partitioned into groups and segments (Sect. 3.1). Section 4 uses the building blocks from the previous sections to present our main MPC perfectly secure protocol with G.O.D. over constant-size rings, with linear communication complexity.

Our work makes use of several functionalities and protocols. To help the reader navigate, we provide in the full version of the paper a list with all of our functionalities and protocols, and their location within the text.

2 Preliminaries

In this work, we focus on functions that can be written as an arithmetic circuit C over the ring $\mathbb{Z}/2^k\mathbb{Z}$ with input, addition, multiplication and output gates. Let $|C|$ denote the size of the circuit C. We will make use of the *client-server* model for secure multiparty computation, in which clients can provide inputs and receive outputs to/from the servers, who are the parties who execute the actual MPC protocol. Note that, if every party plays a single client and a single server, this corresponds to a protocol in the standard MPC model. We assume that every pair of parties, either client and/or server, is connected via a secure (private and authentic) synchronous channel. We measure communication complexity as the total number of bits sent via private channels.[3]

Let c denote the number of clients, n denote the number of servers, and t denote the upper bound of the number of corrupted servers. In this work we focus on the 1/3-corruption setting, *i.e.* $3t + 1 = n$. In this work, we design an MPC protocol where all clients and servers compute the functionality $\mathcal{F}_{\mathsf{Main}}$ with perfect security. Our definition of perfect security is based on the standard simulation-based security which is shown in the work [Can00].

Functionality 1: $\mathcal{F}_{\mathsf{Main}}(C)$

1. Let x denote the input and C denote the circuit. $\mathcal{F}_{\mathsf{Main}}$ receives the input from all clients.
2. $\mathcal{F}_{\mathsf{Main}}$ computes $C(x)$ and distributes the output to all clients.

[3] Since we consider constant-sized rings, this is asymptotically the same as measuring the number of ring elements.

2.1 Party-Elimination Framework

We make use of the party elimination framework by Hirt, Maurer, and Przydatek [HMP00], which constitutes a general strategy to achieve perfect security with G.O.D. with linear communication complexity. The basic idea is to let the parties perform checks that evaluate the correctness of the computation, identifying a pair of parties (with the help of BA for consensus) that contains at least one corrupted party in case of failure; such pair is referred to as a *semi-corrupted pair*. This pair of parties is then *eliminated* (*i.e.* removed from the computation), and the protocol is restarted. To avoid the overhead of re-executing as many times as potential eliminated pairs—which is upper bounded by t—the computation is divided into segments, and the check is performed at the end of each segment. This way, the extra cost of re-running is—in the worst case—t times the cost of each segment, so by keeping segments of appropriate size one can obtain efficient protocols with G.O.D.

We use $\mathcal{P}_{\texttt{active}}$ to denote the set of parties which are active in the current segment, that is, that have not been eliminated. We use $\mathcal{C}_{\texttt{active}} \subset \mathcal{P}_{\texttt{active}}$ for the set of active corrupted parties. Let n' be the size of $\mathcal{P}_{\texttt{active}}$. We use t' for the maximum possible number of the corrupted parties in $\mathcal{P}_{\texttt{active}}$. Each time a semi-corrupted pair is identified, these two parties are removed from $\mathcal{P}_{\texttt{active}}$ and hence $\mathcal{C}_{\texttt{active}}$. It results in $n' := n' - 2$ and $t' := t' - 1$. Initially we have $n = n'$, $t = t'$. Let $T = n' - 2t'$. Therefore, T remains unchanged during the whole protocol.

2.2 Finite Rings

Basic Notation. Let $\mathbb{Z}$ denote the ring of integers. For $q \in \mathbb{Z}$, let $q\mathbb{Z}$ denote the ideal $\{q \cdot n : q \in \mathbb{Z}\}$ and let $\mathbb{Z}/q\mathbb{Z}$ denote the quotient ring, which is the ring of integers modulo q. For a ring $\mathcal{S}$, let $\mathcal{S}[X]$ denote the ring of polynomials in the variable X with coefficients in $\mathcal{S}$. Also, let $\mathcal{S}^*$ denote the multiplicative subgroup of invertible elements in $\mathcal{S}$.

Galois Rings. We adopt the notion of *Galois rings* that contains the quotient ring $\mathbb{Z}/2^k\mathbb{Z}$ from [ACD+19].

Definition 1 (Galois Ring [ACD+19]). *A degree-d* Galois ring *of $\mathbb{Z}/2^k\mathbb{Z}$ is a ring of the form $(\mathbb{Z}/2^k\mathbb{Z})[X]/g(X)$, where k is a positive integer, and $g(X) \in (\mathbb{Z}/2^k\mathbb{Z})[X]$ is a non-constant degree-d polynomial such that its reduction modulo 2 is an irreducible polynomial in the field $\mathbb{F}_2[X]$. We use $\mathsf{GR}(2^k, d)$ to denote degree-d* Galois ring *of $\mathbb{Z}/2^k\mathbb{Z}$.*

In order to interpolate polynomials in a Galois ring, we rely on the following lemma.

Lemma 1 ([ACD+19]). *Let $\mathsf{GR}(2^k, d)$ be a Galois ring with degree d. There exists a length 2^d sequence of distinct elements in $\mathsf{GR}(2^k, d)$ denoted by $\alpha_1, \ldots, \alpha_{2^d}$, such that for any $x_1, \ldots, x_{2^d} \in \mathsf{GR}(2^k, d)$, there exists a unique interpolating polynomial of degree at most $(2^d - 1)$ such that $f(\alpha_i) = x_i$ for all $i \in \{1, 2, \ldots d\}$.*

Using this lemma, we can define necessary components such as Shamir secret sharings and hyper-invertible matrices over Galois rings. In the following, we will use a Galois ring of $\mathbb{Z}/2^k\mathbb{Z}$ denoted by $\mathcal{R} := \mathsf{GR}(2^k, m)$. Note that the size of $\mathcal{R}$ is $2^{m\cdot k}$. We select m such that $2^m \geq 2n+1$ so that it is possible to interpolate degree-$2n$ polynomials in $\mathcal{R}$.

2.3 Secret Sharing Schemes

Shamir Secret Sharing. We will use the standard Shamir secret sharing scheme [Sha79] in this work. For the Galois ring $\mathcal{R} = \mathsf{GR}(2^k, m)$, suppose $(\alpha_i)_{i=1}^n, \beta$ are $n+1$ distinct points, which can be used to interpolate polynomials according to Lemma 1. A degree-d Shamir sharing of $x \in \mathcal{R}$ among $n' \leq n$ parties is a vector $(s_1, s_2, \dots, s_{n'}) \in \mathcal{R}^{n'}$ that satisfies the property that there exists a polynomial $f(\cdot) \in \mathcal{R}^{\leq d}[x]$ with $f(\beta) = x$ and $f(\alpha_i) = s_i, \forall i \in [n']$. The share held by party P_i is s_i. With any $(d+1)$ different shares of the same sharing the secret x can be reconstructed.

A degree-d Shamir sharing of $x \in \mathcal{R}$ is denoted as $[x]_d$. The following two properties hold for Shamir sharings: (1) For all $x, y \in \mathcal{R}$, $[x+y]_d = [x]_d + [y]_d$, and (2) for all $x, y \in \mathcal{R}$ and for all d_1, d_2 subject to $d_1 + d_2 < n$, we have $[x \cdot y]_{d_1+d_2} = [x]_{d_1} \cdot [y]_{d_2}$.

Our protocol also rely heavily on the following property of Shamir sharings. Suppose after some party elimination steps we have n' parties where a maximum t' of them can be malicious.

Lemma 2 ([BTH08]). *Suppose n' parties share a degree-d Shamir sharing $[x]_d$, and at most t' of the shares may be incorrect. If $t' < (n'-d)/2$, then $[x]_d$ is correctable after receiving all the shares, e.g. by Berlekamp-Welch Algorithm. If $t' < n' - d$, then whether $[x]_d$ is inconsistent is detectable after receiving all the shares.*

2.4 Reverse Multiplication Friendly Embeddings

Definition 2 (RMFE over Ring [CRX21,ELXY23]). *Let ℓ, m, k be positive integers. Let $\mathcal{R} = \mathsf{GR}(2^k, m)$ denote the degree-m Galois ring of $\mathbb{Z}/2^k\mathbb{Z}$. A pair of mappings $(\phi : (\mathbb{Z}/2^k\mathbb{Z})^\ell \to \mathcal{R}, \psi : \mathcal{R} \to (\mathbb{Z}/2^k\mathbb{Z})^\ell)$ is called an $(\ell, m)_{2^k}$-reverse multiplication friendly embedding (RMFE) if, for all $\boldsymbol{x}, \boldsymbol{y} \in (\mathbb{Z}/2^k\mathbb{Z})^\ell$, it holds that $\psi(\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{y})) = \boldsymbol{x} \star \boldsymbol{y}$. Without loss of generality we can assume that $\psi(\phi(\mathbf{1})) = 1$, which ensures $\psi(\phi(\boldsymbol{x})) = \boldsymbol{x}$ for all $\boldsymbol{x} \in (\mathbb{Z}/2^k\mathbb{Z})^\ell$.*

Defining the $\mathbb{Z}/2^k\mathbb{Z}$-Linear Map $\mathsf{val}(\cdot)$ [PS21]. To compute the summation of all entries of $\psi(y)$ from $y \in \mathcal{R}$, we define an $\mathbb{Z}/2^k\mathbb{Z}$-linear map $\mathsf{val}(\cdot) : \mathcal{R} \to \mathbb{Z}/2^k\mathbb{Z}$ as follows: For an input y, suppose $\psi(y) = (y_1, y_2, \dots, y_\ell)$, and then $\mathsf{val}(y)$ is defined to be $\sum_{i=1}^\ell y_i$. Let $\boldsymbol{e}_i$ be a vector in $(\mathbb{Z}/2^k\mathbb{Z})^\ell$ such that all entries are 0 except that the i-th entry is 1, and let $\boldsymbol{x}$ be a vector in $(\mathbb{Z}/2^k\mathbb{Z})^\ell$ of which the i-th entry is x_i. According to the definition of RMFEs, we have $\boldsymbol{e}_i \star \boldsymbol{x} = \psi(\phi(\boldsymbol{e}_i) \cdot \phi(\boldsymbol{x}))$. Therefore, we can access x_i by computing $x_i = \mathsf{val}(\phi(\boldsymbol{e}_i) \cdot \phi(\boldsymbol{x}))$.

Existence of Constant Rate RMFEs over Ring $\mathbb{Z}/2^k\mathbb{Z}$ [ACE+21]. In [ACE+21] it has been shown that constant rate RMFEs exist, as summarized in Theorem 1.

Theorem 1. *There exists a family of constant rate $(\ell, m)_{2^k}$-RMFE where $m = \Theta(\ell)$.*

In this work, we will use $(\ell, m)_{2^k}$-RMFE such that $m = O(\log n)$ and $\ell = O(\log n)$. The Galois ring $\mathcal{R} = \mathsf{GR}(2^k, m)$ satisfies $2^m \geq 2n + 1$.

2.5 Useful Building Blocks

Reconstructing Shamir Sharings. The functionality $\mathcal{F}_{\mathsf{OpenPub}}$ takes N degree-d ($d \leq t$) Shamir secret sharings over $\mathcal{R}$ as input, and it outputs the reconstructed secrets to all parties. We assume that for each input degree-d sharing, the shares of all active honest parties lie on a degree-d polynomial.[4] The full description of $\mathcal{F}_{\mathsf{OpenPub}}$ appears in the full version of the paper. An instantiation of this functionality for our ring case can be easily generalized from the field-case construction in [ACD+19], which has communication complexity of $O(N \cdot n \cdot m + n^2 \cdot m)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$.

Secure Multiplication. The functionality $\mathcal{F}_{\mathsf{Mult}}$ takes two tuples of N degree-t Shamir sharings over $\mathcal{R}$ as input, which are denoted by $([x_1]_t, \ldots, [x_N]_t)$ and $([y_1]_t, \ldots, [y_N]_t)$. The output of $\mathcal{F}_{\mathsf{Mult}}$ is the tuple of degree-t Shamir sharings of the results $([x_1 \cdot y_1]_t, \ldots, [x_N \cdot y_N]_t)$. We assume that for each input degree-t Shamir sharing, the shares of all active honest parties lie on a degree-t polynomial. The full description of $\mathcal{F}_{\mathsf{Mult}}$ appears in the full version of the paper. An instantiation of this functionality for our ring case can be easily generalized from the field-case construction in [BTH08]. Also, another instantiation of this functionality is implied in [ACD+19]. The protocol generalized from [BTH08] has communication complexity of $O(N \cdot n \cdot m + n^2 \cdot m \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$, where S is the set of eliminated parties.

Performing Re-encode. In our construction, we will need to transform a degree-t Shamir sharing over $\mathcal{R}$ from $[x]_t$ to $[\phi \circ \psi(x)]_t$ in order to evaluate multiplication gates in the circuit. This process is called *re-encode*. The functionality $\mathcal{F}_{\mathsf{ReEncode}}$ takes N degree-t Shamir sharings over $\mathcal{R}$ as input, which are denoted by $[x_1]_t, \ldots, [x_N]_t$. The output of $\mathcal{F}_{\mathsf{ReEncode}}$ are N degree-t Shamir sharings of the re-encoded result $[\phi \circ \psi(x_1)]_t, \ldots, [\phi \circ \psi(x_N)]_t$. We assume that for each input degree-t Shamir sharing, the shares of all active honest parties lie on a degree-t polynomial. The full description of $\mathcal{F}_{\mathsf{ReEncode}}$ appears in the full version of the paper. An instantiation of this functionality for our ring case can be easily

[4] If this is not the case, we ask the functionality to send the active honest parties' inputs to the adversary and allow the adversary to decide the output of active honest parties. Essentially, we give up the security if the shares of active honest parties do not lie on degree-d polynomials.

generalized from the field-case construction in [CCXY18]. The instantiation has communication complexity of $O(N \cdot n \cdot m + n^2 \cdot m \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$, where S is the set of eliminated parties.

Verifying Consistency of Unreliable Broadcast Values. In our protocol, we will ask a dealer D to distribute several values that are supposed to be all the same, towards all parties. We do this over point-to-point channel to save the communication. The functionality $\mathcal{F}_{\mathsf{VerifyBC}}$ receives from all parties N such unreliable broadcast values dealt by a dealer D, and verifies whether all parties indeed received the same values. The output of this functionality to each party is either `consistent` or $(\texttt{inconsistent}, E)$, where E is a semi-corrupted pair of parties. The full description of $\mathcal{F}_{\mathsf{VerifyBC}}$ appears in the full version of the paper. An instantiation of this functionality for our ring case can be easily generalized from the field-case construction in [BTH08], which has communication complexity of $O(N \cdot n \cdot m + n^2 \cdot m)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$.

Input Gates for Shamir Sharings. We introduce the functionality $\mathcal{F}_{\mathsf{InputShamir}}$, where a client Client with N inputs in $\mathcal{R}$ shares its inputs to the active parties using Shamir secret sharing. The full description of $\mathcal{F}_{\mathsf{InputShamir}}$ appears in the full version of the paper. An instantiation of this functionality for our ring case can be easily generalized from the field-case construction in [BTH08]. The instantiation has a communication complexity of $O(N \cdot n \cdot m + n^2 \cdot m \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$, where S is the set of eliminated parties.

2.6 Preparing Correlated Randomness

Our protocol relies on different forms of correlated randomness shared by all parties, and these are prepared independently of the inputs of the clients. We give a brief description of the correlations we require below.

Random Shamir Sharings. The functionality $\mathcal{F}_{\mathsf{RandShamir}}$ enables all parties to prepare N random degree-t Shamir sharings in the form of $[\phi(\boldsymbol{r})]_t$, where $\boldsymbol{r}$ is a random vector in $(\mathbb{Z}/2^k\mathbb{Z})^\ell$. The description and the instantiation of $\mathcal{F}_{\mathsf{RandShamir}}$ can be found in the full version of the paper. The total communication complexity for the instantiation of $\mathcal{F}_{\mathsf{RandShamir}}$ to generate N random Shamir sharings is $O(N \cdot n \cdot m + n^2 \cdot m^2 \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$, where S is the set of eliminated parties.

Random Zero Additive Sharings. Once Beaver triples are prepared in the preprocessing phase, parties only need to do reconstructions in the online phase. To protect the shares held by honest parties, for each reconstruction, we will prepare a random additive sharing of 0 among the first $t+1$ parties. The functionality $\mathcal{F}_{\mathsf{RandZeroAdditive}}$ enables all parties to prepare N random zero additive sharings. The description and the instantiation of $\mathcal{F}_{\mathsf{RandZeroAdd}}$ can be found in the full version of the paper. The total communication complexity for the instantiation of $\mathcal{F}_{\mathsf{RandZeroAdd}}$ to generate N zero additive sharings is $O(N \cdot n \cdot m + n^2 \cdot m \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$, where S is the set of eliminated parties.

Random Parity Sharings. For an element $p \in \mathcal{R}$, we say that p is *parity element* if $\mathsf{val}(p) = 0$, and a parity sharing is a degree-t Shamir sharing of a parity element. When localizing a fault within a circuit segment, uniformly random parity sharings will be used as masks so that it is possible to check the correctness of the reconstruction. The functionality $\mathcal{F}_{\mathsf{RandParity}}$ enables all parties to prepare N random parity sharings. The description and instantiation of $\mathcal{F}_{\mathsf{RandParity}}$ can be found in the full version of the paper. The total communication complexity for the instantiation of $\mathcal{F}_{\mathsf{RandParity}}$ to generate N random parity sharings is $O(N \cdot n \cdot m + n^2 \cdot m^2 \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$, where S is the set of eliminated parties.

Beaver Triples. To evaluate addition gates and multiplication gates, all parties will prepare Beaver triples in the form of $([\phi(\boldsymbol{a})]_t, [\phi(\boldsymbol{b})]_t, [\phi(\boldsymbol{c})]_t)$, where $\boldsymbol{a} + \boldsymbol{b} = \boldsymbol{c}$ when the Beaver triple is additive and $\boldsymbol{a} \star \boldsymbol{b} = \boldsymbol{c}$ when the Beaver triple is multiplicative. We introduce two functionalities $\mathcal{F}_{\mathsf{TripleAdd}}$ and $\mathcal{F}_{\mathsf{TripleMult}}$. $\mathcal{F}_{\mathsf{TripleAdd}}$ enables all parties to prepare N random additive Beaver triples, and $\mathcal{F}_{\mathsf{TripleMult}}$ enables all parties to prepare N random multiplicative Beaver triples. The descriptions and the instantiations of both $\mathcal{F}_{\mathsf{TripleAdd}}$ and $\mathcal{F}_{\mathsf{TripleMult}}$ can be found in the full version of the paper. The total communication complexity for the instantiation of either $\mathcal{F}_{\mathsf{TripleAdd}}$ or $\mathcal{F}_{\mathsf{TripleMult}}$ to generate N Beaver Triples is $O(N \cdot n \cdot m + n^2 \cdot m^2 \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$, where S is the set of eliminated parties.

3 Segment Evaluation and Verification

Our protocol first splits the circuit into *segments*, and then assigns gates of each type within a segment into *gate groups*. This is discussed in Sect. 3.1. Then, in Sect. 3.2 we show how the parties evaluate the gates of a given circuit *optimistically*, that is, without checking that the computation was carried out correctly. The verification is discussed in Sect. 3.3.

3.1 Groups and Segments

We need to split the circuit into n segments in order to apply packing effectively. We assume that the circuit C satisfies the following conditions:

1. **Circuit Segment Conditions:**
   - Because C is a Directed Acyclic Graph (DAG), there exists a topological ordering among all addition and multiplication gates. We require that each segment consists of addition and multiplication gates whose topological orders are consecutive.
   - The size of each segment should be $O(n \cdot m^2 + |C|/n)$.
2. **Gate Number Conditions:**
   - In the input and output layers, the number of input gates belonging to each client and the number of output gates belonging to each client are multiples of ℓ.

   - The number of addition and multiplication gates within each circuit segment are multiples of ℓ.
3. **Gate Grouping Conditions:**
   - During the computation, gates that have the same type (i.e., input gates belonging to the same client, output gates belonging to the same client, multiplication gates in the same circuit segment, addition gates in the same circuit segment) are organized into gate groups of size ℓ.
   - For the output wires of each gate group, the number of times that those wires are used as input wires in other gates is a multiple of ℓ.

In the full version of the paper we show that, if C does not satisfy these properties, then it can be transformed into a circuit C' that does satisfy the properties without affecting our linear communication claim. Based on the conditions above, we can split each segment into *gate groups* consisting of either ℓ multiplication gates (in which case the group is a *multiplication group*), or ℓ addition gates (in which case the group is an *addition group*). A set of ℓ wires corresponding to the left or right inputs of a given gate group is referred to as an *input group*, and *output groups* are defined similarly, but with output wires. The transformed circuit has size $|C'| = O(|C| + \ell \cdot c + n^2 \cdot m^2)$.

3.2 Segment Evaluation

The focus of this section is to show how the parties can evaluate optimistically a given segment $\mathtt{seg}$. The overall idea is to use additive secret-sharing with multiplication triples derived from packed triples. However, we consider an "enhanced" version of additive secret-sharing that allows for fault detection in case of cheating. This is described below.

Extended Additive Sharings. Let (ϕ, ψ) be an $(\ell, m)_2$-RMFE. Recall that n denotes the number of parties and $\phi : (\mathbb{Z}/2^k\mathbb{Z})^\ell \to \mathcal{R}$ is an $\mathbb{Z}/2^k\mathbb{Z}$-linear map. Also, $\mathcal{R} = \mathsf{GR}(2^k, m)$ such that $2^m \geq 2n + 1$. Therefore, Shamir secret sharing is well-defined in $\mathcal{R}$. In our construction, we will use ϕ to encode a vector of secrets $\boldsymbol{x} = (x^{(1)}, x^{(2)}, \dots, x^{(\ell)}) \in (\mathbb{Z}/2^k\mathbb{Z})^\ell$. All parties will hold a degree-t Shamir sharing of $\phi(\boldsymbol{x})$, denoted by $[\phi(\boldsymbol{x})]_t$. For $x \in \mathbb{Z}/2^k\mathbb{Z}$, we use $\langle x \rangle$ to denote an *additive sharing* of x among the first $t+1$ parties in $\mathbb{Z}/2^k\mathbb{Z}$. Recall that n' denotes the number of remaining parties after the previous party elimination steps. Specifically, the additive sharing of x is $\langle x \rangle = (x_1, \dots, x_{n'})$ where party P_i holds the share $x_i \in \mathbb{Z}/2^k\mathbb{Z}$ such that $\sum_{i=1}^{t+1} x_i = x$ and $x_{t+2}, \dots, x_{n'}$ are all 0. Recall that $\psi : \mathcal{R} \to (\mathbb{Z}/2^k\mathbb{Z})^\ell$ and $\mathsf{val}(\cdot) : \mathcal{R} \to \mathbb{Z}/2^k\mathbb{Z}$ are both $\mathbb{Z}/2^k\mathbb{Z}$-linear. The parties have *extended additive sharings* of $x \in \mathbb{Z}/2^k\mathbb{Z}$, denoted by $(\!|x|\!)$, if they have a degree-t Shamir sharing $[y]_t$ in $\mathcal{R}$ such that $\mathsf{val}(y) = x$. We write $(\!|x|\!) := [y]_t$. It is clear that these sharings are additive.

We note that we can derive extended additive sharings of $x \in \mathbb{Z}/2^k\mathbb{Z}$ from Shamir sharings $[\phi(\boldsymbol{z})]_t$, where the j-th element of $\boldsymbol{z}$ is x. To see this, observe that, by the property of RMFE, we have that $\psi(\phi(\boldsymbol{e}_j) \cdot \phi(\boldsymbol{z})) = \boldsymbol{e}_j \star \boldsymbol{z}$. Therefore,

$\mathsf{val}(\phi(\boldsymbol{e}_j) \cdot \phi(\boldsymbol{z})) = x$. To obtain $(\!|x|\!)$, all parties locally compute $(\!|x|\!) = \phi(\boldsymbol{e}_j) \cdot [\boldsymbol{z}]_t$. In addition, it is easy to obtain sharings $\langle x \rangle$ from $(\!|x|\!)$ by using Lagrange coefficients; we give the details in the full version of the paper, where we describe at length the notion of extended sharings, together with their properties.

Optimistically Evaluating a Segment. For a circuit segment $\mathtt{seg}$, we use Protocol $\Pi_{\mathsf{Eval}}(\mathtt{seg})$ to optimistically evaluate this segment. We evaluate its addition and multiplication gates using extended additive sharings and Beaver triples. A Beaver triple $([\phi(\boldsymbol{a})]_t, [\phi(\boldsymbol{b})]_t, [\phi(\boldsymbol{c})]_t)$ can be used to evaluate ℓ addition gates or ℓ multiplication gates. We assume that all parties have computed Shamir sharings of all gate group outputs of the previous circuit segments. This means that for an i-th element of any $[\phi(\boldsymbol{z})]_t$ used as a gate input in $\mathtt{seg}$, all parties can locally compute the extended additive sharing $(\!|x|\!) := \phi(\boldsymbol{e}_i) \cdot [\phi(\boldsymbol{z})]_t$.

For each gate with extended additive input sharings $(\!|x|\!), (\!|y|\!)$, we will use the Beaver triple associated to the gate group containing this gate. Suppose the gate is the j-th gate within the gate group, and suppose $([\phi(\boldsymbol{a})]_t, [\phi(\boldsymbol{b})]_t, [\phi(\boldsymbol{c})]_t)$ is the Beaver triple corresponding to the gate group. All parties compute the extended additive sharings $(\!|a_j|\!) := \phi(\boldsymbol{e}_j) \cdot [\phi(\boldsymbol{a})]_t$, $(\!|b_j|\!) := \phi(\boldsymbol{e}_j) \cdot [\phi(\boldsymbol{b})]_t$ and $(\!|c_j|\!) := \phi(\boldsymbol{e}_j) \cdot [\phi(\boldsymbol{c})]_t$. Then all parties derive the additive sharings $\langle x \rangle$, $\langle y \rangle$, $\langle a_j \rangle$, $\langle b_j \rangle$ from the extended additive sharings using the method described previously.

The next step is reconstructing $\langle x \rangle + \langle a_j \rangle$ and $\langle y \rangle + \langle b_j \rangle$, for which a fixed dealer D will receive all shares, and then sends the reconstructed value to all parties. However, a subtle issue is that all parties must protect the redundancy in their sharings by preparing two random zero additive sharing $\langle o_1 \rangle, \langle o_2 \rangle$, which can be done with Functionality $\mathcal{F}_{\mathsf{RandZeroAdd}}$. Then, all parties send their shares of $\langle x \rangle + \langle a_j \rangle + \langle o_1 \rangle$ and $\langle y \rangle + \langle b_j \rangle + \langle o_2 \rangle$ to the dealer D, who reconstructs $u := x + a_j$ and $v := y + b_j$, and sends the result to all other parties.

If the gate is an addition gate, all parties locally compute the output extended additive sharing $(\!|z|\!) := (u + v) \cdot \phi(\boldsymbol{e}_j) - (\!|c_j|\!)$. If the gate is a multiplication gate, all parties locally compute the output extended additive sharing $(\!|z|\!) := (u \cdot v) \cdot \phi(\boldsymbol{e}_j) - u \cdot (\!|b_j|\!) - v \cdot (\!|a_j|\!) + (\!|c_j|\!)$. We describe Π_{Eval} in full detail below, and we show that the communication cost of Π_{Eval} is $O(n^3 \cdot m^2)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$.

Protocol 1: $\Pi_{\mathsf{Eval}}(\mathtt{seg})$

1. Suppose $\mathtt{seg}$ has $n_{\mathrm{add}} \cdot \ell$ addition gates and $n_{\mathrm{mult}} \cdot \ell$ multiplication gates. All parties select the active party with the smallest index as the dealer of this segment $\mathtt{seg}$. Let D denote the dealer.
2. All parties call $\mathcal{F}_{\mathsf{RandZeroAdd}}$ to prepare $2 \cdot (n_{\mathrm{add}} + n_{\mathrm{mult}}) \cdot \ell$ random additive zero sharings. All parties also receive a set of eliminated parties denoted by S_1. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_1$.
3. All parties call $\mathcal{F}_{\mathsf{TripleAdd}}(n_{\mathrm{add}} \cdot \ell)$ to generate the additive Beaver triples for the segments. All parties receive a set of eliminated parties denoted by S_2. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_2$.
   All parties call $\mathcal{F}_{\mathsf{TripleMult}}(n_{\mathrm{mult}} \cdot \ell)$ to generate the multiplicative Beaver

triples for the segments. All parties receive a set of eliminated parties denoted by S_3. All parties update $\mathcal{P}_{\texttt{active}} := \mathcal{P}_{\texttt{active}} - S_3$.

4. All parties locally get all the extended additive sharings for the gate inputs of `seg` that are collected from previous layers. For the gate input connected to the i-th wire of the Shamir sharing $[\phi(\boldsymbol{z})]_t$, all parties locally derive the extended additive sharing by $\phi(\boldsymbol{e}_i) \cdot [\phi(\boldsymbol{z})]_t$.
5. All parties evaluate the multiplication gates and addition gates within `seg` according to topological ordering. For each gate with input extended additive sharings $(\!|x|\!)$ and $(\!|y|\!)$, we suppose it corresponds to the j-th entry of the Beaver triple $([\phi(\boldsymbol{a})]_t, [\phi(\boldsymbol{b})]_t, [\phi(\boldsymbol{c})]_t)$, denoted by (a_j, b_j, c_j). All parties consume two unused random additive sharings prepared in Step 2, denoted by $\langle o_1 \rangle$ and $\langle o_2 \rangle$. Then all parties perform the following steps:
   (a) All parties locally derive the extended additive sharing for a_j, b_j, c_j with $(\!|a_j|\!) = \phi(\boldsymbol{e}_j)\cdot[\phi(\boldsymbol{a})]_t$, $(\!|b_j|\!) = \phi(\boldsymbol{e}_j)\cdot[\phi(\boldsymbol{b})]_t$ and $(\!|c_j|\!) = \phi(\boldsymbol{e}_j)\cdot[\phi(\boldsymbol{c})]_t$.
   (b) All parties locally computes $\langle x \rangle + \langle a_j \rangle + \langle o_1 \rangle$ and $\langle y \rangle + \langle b_j \rangle + \langle o_2 \rangle$, and send their shares to D.
   (c) D reconstructs $u := x + a_j$ and $v := y + b_j$. Then D sends u and v to all parties.
   (d) If the gate is an addition gate, all parties locally compute the output extended additive sharing $(\!|z|\!) := (u + v) \cdot \phi(\boldsymbol{e}_j) - (\!|c_j|\!)$.
   If the gate is a multiplication gate, all parties locally compute the output extended additive sharing $(\!|z|\!) := (u \cdot v) \cdot \phi(\boldsymbol{e}_j) - u \cdot (\!|b_j|\!) - v \cdot (\!|a_j|\!) + (\!|c_j|\!)$.
6. All parties output the eliminated set of parties $S := S_1 \cup S_2 \cup S_3$.

Cost of Π_{Eval}. Recall that each circuit segment has $O(|C|/n + n^2 \cdot m^2)$ gates, so we have $n_{\text{add}} = O(|C|/(n \cdot \ell) + n^2 \cdot m^2/\ell)$ and $n_{\text{mult}} = O(|C|/(n \cdot \ell) + n^2 \cdot m^2/\ell)$. It follows that the communication complexity of Step 2, Step 3, and Step 5 are all $O(|C| + n^2 \cdot m^2 \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$. Therefore, the communication complexity of Π_{Eval} is $O(|C| + n^2 \cdot m^2 \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$.

3.3 A First (Inefficient) Verification Protocol

For a given segment `seg`, once the parties have executed Π_{Eval}(`seg`), they have obtained extended additive sharings of every intermediate wire value of the circuit. However, a corrupted party may have cheated during this protocol, perhaps by sending incorrect sharings to the dealer D, and/or D itself sends incorrect reconstructions to the parties. The purpose of this section is discussing how the parties can check whether such cheating indeed took place or not. The parties do this before they proceed to the next segment evaluation. Our strategy, as outlined in Sect. 1.3, is to get Shamir sharings of all input groups in `seg`, which will be used to verify that the openings are done correctly. These sharings are derived from Shamir sharings of the output groups in the previous segments, and possible of `seg` itself. In order to achieve this, we must apply *network routing* techniques, which are developed in the works of [GPS21, GPS22].

We begin by presenting a version of our verification protocol that does *not* yet satisfy the linear communication complexity claim, but is structurally very close to our actual protocol while requiring little preliminaries on network routing for a clear understanding.

Network Routing. Consider a segment $\texttt{seg}$, and suppose that the parties have sharings $[\phi(\boldsymbol{z})]_t$ for every output group $\boldsymbol{z}$ of each segment prior to $\texttt{seg}$, and also of $\texttt{seg}$ itself. Now, let $\boldsymbol{x}$ be an input group in $\texttt{seg}$. Each entry x_i in $\boldsymbol{x}$ is the output of a previous gate, which appears in an output group of either $\texttt{seg}$, or a prior segment. Network routing is a set of techniques that enables the parties to obtain, from the sharings of all previous groups $[\phi(\boldsymbol{z})]_t$, sharings $[\phi(\boldsymbol{x})]_t$ of the input group $\boldsymbol{x}$ in $\texttt{seg}$. This is crucial for our verification protocol, and it is achieved by Protocol $\mathit{\Pi}_{\mathsf{NetworkRouting}}(\texttt{seg})$ (Protocol 2 in p. 24). The construction of $\mathit{\Pi}_{\mathsf{NetworkRouting}}(\texttt{seg})$ is a natural adaptation to the RMFE setting of the network routing techniques from [GPS21, GPS22], which are originally set in the packed secret-sharing context.

For our first verification protocol, we will use $\mathit{\Pi}_{\mathsf{NetworkRouting}}$ as a "black-box". Doing so results in a verification protocol with super-linear communication, stemming from the fact that all the calls to $\mathit{\Pi}_{\mathsf{NetworkRouting}}(\texttt{seg})$ across all segments $\texttt{seg}$ redo a lot of computation that can be done only once if one "opens the box". Later in the section we dig into the details of $\mathit{\Pi}_{\mathsf{NetworkRouting}}(\texttt{seg})$, identifying these steps that are repeated across calls so that they are called only once, avoiding unnecessary repetition and hence achieving the desired linear communication complexity.

Fault Detection. We need to introduce one more tool before we present our first (inefficient) verification protocol. Our verification ultimately boils down to ensuring that extended secret-shared values, that have been opened non-robustly using additive shares through a dealer, have actually been opened correctly. If this is not the case, the parties should be able to identify a semi-corrupted pair. This is achieved by means of a protocol $\mathit{\Pi}_{\mathsf{FaultDetection}}$ that takes as input an inconsistent pair of extended additive sharing $(\!|s|\!)$ and the masked additive sharing $\langle s\rangle + \langle o\rangle$ corresponding to it. At a high level, in this protocol the dealer will open the shares of the extended additive sharing and the masked additive sharing to find out a dispute between parties. We refer the readers to Sect. 3.6 for more details.

Inefficient Verification. Consider a segment $\texttt{seg}$. Let $([\phi(\boldsymbol{a}^{(i)})]_t, [\phi(\boldsymbol{b}^{(i)})]_t, [\phi(\boldsymbol{c}^{(i)})]_t)$ denote the Beaver triple corresponding to the i-th gate group in $\texttt{seg}$, and let $\boldsymbol{x}^{(i)}$ and $\boldsymbol{y}^{(i)}$ denote the left and right inputs of the gate group. Note that before the verification of $\texttt{seg}$, each party locally holds the values $u_j^{(i)} = x_j^{(i)} + a_j^{(i)}$ and $v_j^{(i)} = y_j^{(i)} + b_j^{(i)}$ sent by D for $i \in [N], j \in [\ell]$. All parties first check the consistency of the values sent by D by calling $\mathcal{F}_{\mathsf{VerifyBC}}$. If the values are consistent, all parties can get "temporary" input group sharings $[\phi(\boldsymbol{x})]_t$ and $[\phi(\boldsymbol{y})]_t$ from with $\boldsymbol{u}^{(i)}$, $\boldsymbol{v}^{(i)}$ and the Beaver triple $([\phi(\boldsymbol{a}^{(i)})]_t, [\phi(\boldsymbol{b}^{(i)})]_t, [\phi(\boldsymbol{c}^{(i)})]_t)$, and they use $\mathcal{F}_{\mathsf{Mult}}$ and $\mathcal{F}_{\mathsf{ReEncode}}$ to obtain $[\phi(\boldsymbol{x}^{(i)} \star \boldsymbol{y}^{(i)})]_t$.

After getting all output Shamir sharings in seg, the parties use $\Pi_{\mathsf{NetworkRouting}}$ to obtain Shamir sharings of all the input groups $[\phi(\boldsymbol{x}^{(i)})]_t$ and $[\phi(\boldsymbol{y}^{(i)})]_t$. The parties robustly reconstruct $[\phi(\boldsymbol{x}^{(i)} + \boldsymbol{a}^{(i)})]_t$ and $[\phi(\boldsymbol{y}^{(i)} + \boldsymbol{b}^{(i)})]_t$ using $\mathcal{F}_{\mathsf{OpenPub}}$, and then they compare these outputs with the values reconstructed in the evaluation phase $\boldsymbol{u}^{(i)}$ and $\boldsymbol{v}^{(i)}$. If all entries are consistent, the evaluation of seg is correct. Otherwise, all parties can locate the first inconsistent $x_j^{(i)} + a_j^{(i)}$ or $y_j^{(i)} + b_j^{(i)}$. Then they eliminate a set of semi-corrupted parties, and re-evaluate the circuit segment afterwards.

The steps of the verification protocol are the following. We reiterate that this does not have linear communication complexity, but only because of the calls to $\Pi_{\mathsf{NetworkRouting}}$. This is addressed in Sect. 3.5 after we "open the box" of network routing in Sect. 3.4

1. Suppose seg has N gate groups. For the i-th gate group within seg that corresponds to the Beaver triple $([\phi(\boldsymbol{a}^{(i)})]_t, [\phi(\boldsymbol{b}^{(i)})]_t, [\phi(\boldsymbol{c}^{(i)})]_t)$ and has inputs $\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \in (\mathbb{Z}/2^k\mathbb{Z})^\ell$ All parties locally compute $\phi(\boldsymbol{x}^{(i)}+\boldsymbol{a}^{(i)})$ and $\phi(\boldsymbol{y}^{(i)}+\boldsymbol{b}^{(i)})$ for all $i \in [N]$.
2. All parties call $\mathcal{F}_{\mathsf{VerifyBC}}$ with inputs $\{\phi(\boldsymbol{x}^{(i)}+\boldsymbol{a}^{(i)})\}_{i\in[N]} \cup \{\phi(\boldsymbol{y}^{(i)}+\boldsymbol{b}^{(i)})\}_{i\in[N]}$ and D. If the result is a semi-honest pair $\{P_{j_1}, P_{j_2}\}$, all parties take it as output and halt. Otherwise, run the following steps.
3. All parties locally compute $[\phi(\boldsymbol{x}^{(i)})]_t = \phi(\boldsymbol{x}^{(i)} + \boldsymbol{a}^{(i)}) - [\phi(\boldsymbol{a}^{(i)})]_t$ and $[\phi(\boldsymbol{y}^{(i)})]_t = \phi(\boldsymbol{y}^{(i)} + \boldsymbol{b}^{(i)}) - [\phi(\boldsymbol{b}^{(i)})]_t$ for all $i \in [N]$.
4. For each additive gate group with input $[\phi(\boldsymbol{x}^{(i)})]_t$ and $[\phi(\boldsymbol{y}^{(i)})]_t$, all parties locally compute the output sharing $[\phi(\boldsymbol{z}^{(i)})]_t = [\phi(\boldsymbol{x}^{(i)})]_t + [\phi(\boldsymbol{y}^{(i)})]_t$. For all the multiplicative gate groups, suppose their indices form the set I_{mult}. All parties call $\mathcal{F}_{\mathsf{Mult}}$ with input $([\phi(\boldsymbol{x}^{(i)})]_t)_{i\in I_{\text{mult}}}$ and $([\phi(\boldsymbol{y}^{(i)})]_t)_{i\in I_{\text{mult}}}$, and get output $([w^{(i)}]_t)_{i\in I_{\text{mult}}}$ and a set of eliminated parties denoted by S_1. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_1$.
   Then all the parties call the functionality $\mathcal{F}_{\mathsf{ReEncode}}$ with input $([w^{(i)}]_t)_{i\in I_{\text{mult}}}$ and get the output sharings $([\phi(\boldsymbol{z}^{(i)})]_t)_{i\in I_{\text{mult}}}$ and a set of eliminated parties denoted by S_2. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_2$.
5. All parties run the protocol $\Pi_{\mathsf{NetworkRouting}}(\mathtt{seg})$ to get all Shamir sharings of seg's gate group inputs, denoted by $\{[\phi(\tilde{\boldsymbol{x}}^{(i)})]_t\}_{i\in[N]}$ and $\{[\phi(\tilde{\boldsymbol{y}}^{(i)})]_t\}_{i\in[N]}$. All parties also receive a set of eliminated parties denoted by S_3. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_3$.
6. All parties call $\mathcal{F}_{\mathsf{OpenPub}}$ to reconstruct the Shamir sharings $\{[\phi(\tilde{\boldsymbol{x}}^{(i)})]_t + [\phi(\boldsymbol{a}^{(i)})]_t\}_{i\in[N]}$ and $\{[\phi(\tilde{\boldsymbol{y}}^{(i)})]_t + [\phi(\boldsymbol{b}^{(i)})]_t\}_{i\in[N]}$, and get $\tilde{\boldsymbol{x}}^{(i)} + \boldsymbol{a}^{(i)}$ and $\tilde{\boldsymbol{y}}^{(i)} + \boldsymbol{b}^{(i)}$ for all $i \in [N]$.
7. Each party locally compares $\tilde{\boldsymbol{x}}^{(i)}+\boldsymbol{a}^{(i)}$ with $\boldsymbol{x}^{(i)}+\boldsymbol{a}^{(i)}$ and compares $\tilde{\boldsymbol{y}}^{(i)}+\boldsymbol{b}^{(i)}$ with $\boldsymbol{y}^{(i)} + \boldsymbol{b}^{(i)}$ for all $i \in [N]$. If there are any differences, all parties do the following: let $S := S_0 \cup S_1 \cup S_2 \cup S_3$. If $S \neq \emptyset$, all parties output S and incorrect and halts. Otherwise, all parties can select a wire with inconsistent value that has the smallest topological order, denoted by $x_{j_0}^{(i_0)} + a_{j_0}^{(i_0)}$ or $y_{j_0}^{(i_0)} + b_{j_0}^{(i_0)}$. Let $\langle s\rangle + \langle o\rangle$ denote its corresponding additive sharing for reconstruction in the protocol Π_{Eval}, and let $(\!|s|\!)$ denote its extended additive sharing. Then

all parties run the protocol $\Pi_{\mathsf{FaultDetection}}(D, \langle s \rangle + \langle o \rangle, (\!| s |\!))$, and get a set of eliminated parties S_4 as output. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_4$, and output $S \cup S_4$ and `incorrect`.

3.4 Details on Network Routing

In this section we describe in detail how network routing works in order to identify the pieces that can be re-used from one call to $\Pi_{\mathsf{NetworkRouting}}$ to the next, and then we present our actual verification protocol with linear communication in Sect. 3.5.

Fan-Out Operations. The first ingredient of network routing is a functionality $\mathcal{F}_{\mathsf{FanOut}}$, which we describe in detail as Functionality ?? in the full version of the paper. $\mathcal{F}_{\mathsf{FanOut}}$ takes as input a list of sharings $[\phi(\boldsymbol{z}^{(1)})]_t, \dots, [\phi(\boldsymbol{z}^{(N)})]_t$, and also $n_j^{(i)} \in \mathbb{Z}^+$ for every $i \in [N]$ and $j \in [\ell]$ where ℓ divides $\sum_{j=1}^{\ell} n_j^{(i)}$. The functionality outputs, for each $i \in [N]$, sharings $[\phi(\boldsymbol{x}_j^{(i)})]_t$ for $j \in [m^{(i)}]$ where $m^{(i)} := \left(\sum_{j=1}^{\ell} n_j^{(i)}\right)/\ell$, such that each $z_j^{(i)}$ appears exactly $n_j^{(i)}$ times in $[\boldsymbol{x}_1^{(i)} \| \cdots \| \boldsymbol{x}_{m^{(i)}}^{(i)}]$. Jumping ahead, fan-out will be used to copy each output wire as many times as it is used subsequently in the circuit.

The protocol Π_{FanOut} that implements $\mathcal{F}_{\mathsf{FanOut}}$ can be found in the full version of the paper, and its communication complexity of the protocol to generate a total of M fan-out sharings is $O(M \cdot n \cdot m + n^2 \cdot m^2 \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$, where S is the set of eliminated parties.

Secret Collection. The second crucial operation that is needed in network routing is, given a series of Shamir sharings $([\phi(\boldsymbol{x}_1)]_t, \dots, [\phi(\boldsymbol{x}_N)]_t)$, obtain another set of sharings $([\phi(\boldsymbol{y}_1)]_t, \dots, [\phi(\boldsymbol{y}_N)]_t)$, where $(\boldsymbol{y}_1 \| \cdots \| \boldsymbol{y}_N)$ is a permutation of $(\boldsymbol{x}_1 \| \cdots \| \boldsymbol{x}_N)$. We refer to this operation as *secret collection.* To gain some intuition on how this helps in network routing, suppose that $\mathcal{F}_{\mathsf{FanOut}}$ has been applied to all wires in the circuit, copying them as many times as they appear in future gates, and let $\boldsymbol{x} = (\boldsymbol{x}_1, \dots, \boldsymbol{x}_N)$ be the all the vectors output by $\mathcal{F}_{\mathsf{FanOut}}$. Let $\boldsymbol{y} = (\boldsymbol{y}_1, \dots, \boldsymbol{y}_N)$ consist of all input groups to all circuit segments, and also to the output layer. We have then that $\boldsymbol{y}$ is a *permutation* of $\boldsymbol{x}$. Using secret collection, the parties can obtain sharings of each input group $[\phi(\boldsymbol{y}_i)]_t$, which is precisely what is needed for network routing.

The following theorem from [GPS21] is useful for implementing this secret collection functionality.

Theorem 2 ([GPS21]). *Suppose $\boldsymbol{x} = (\boldsymbol{x}_1, \dots, \boldsymbol{x}_N)$ and $\boldsymbol{y} = (\boldsymbol{y}_1, \dots, \boldsymbol{y}_N)$ satisfy $P \cdot \boldsymbol{x} = \boldsymbol{y}$ where P is a permutation matrix. There exists two sets of permutations $p_1, \dots, p_N$ and $q_1, \dots, q_N$ that permute vectors in $(\mathbb{Z}/2^k\mathbb{Z})^\ell$, such that after applying p_i to $\boldsymbol{x}_i$ and q_j to $\boldsymbol{y}_j$ for all $i, j \in [N]$, the following property holds for an arbitrary $q_h \cdot \boldsymbol{y}_h$:*

- *Suppose that $q_h \cdot \boldsymbol{y}_h = (y'_1, \dots, y'_\ell)$. Then, for all $w \in [\ell]$, there exists $v_{h,w} \in [\ell]$ such that y'_w is equal to the w-th entry in $p_{v_{h,w}} \cdot \boldsymbol{x}_{v_{h,w}}$.*

Following this theorem, we can obtain $\boldsymbol{y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N)$ by first permuting each $[\phi(\boldsymbol{x}_i)]_t$ as $[\phi(p_i \cdot \boldsymbol{x}_i)]_t$. Then, for each $h \in [N]$ parties compute locally $\sum_{w=1}^{\ell} \phi(\boldsymbol{e}_w) \cdot [\phi(p_{v_{h,w}} \cdot \boldsymbol{x}_{v_{h,w}})]_t$, which creates a vector whose w-th entry is the w-th entry of $p_{v_{h,w}} \cdot \boldsymbol{x}_{v_{h,w}}$. Thanks to the theorem, this is precisely $q_h \cdot \boldsymbol{y}_h$, so the parties can obtain $[\phi(q_h \cdot \boldsymbol{y}_h)]_t$ by applying $\mathcal{F}_{\mathsf{ReEncode}}$. Finally, to obtain the desired $[\phi(\boldsymbol{y}_h)]_t$ for each $h \in [N]$, the parties can apply the inverse permutation q_h^{-1} of q_h to the sharing $[\phi(q_h \cdot \boldsymbol{y}_h)]_t$.

The permutations above are done with a functionality $\mathcal{F}_{\mathsf{Permute}}$, which we define and instantiate with Protocol Π_{Permute} in the full version of the paper, involving a communication complexity of $O(N \cdot n \cdot m + n^2 \cdot m^2 \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$, where S is the set of eliminated parties. The aggregation $\sum_{w=1}^{\ell} \phi(\boldsymbol{e}_w) \cdot [\phi(p_{v_{h,w}} \cdot \boldsymbol{x}_{v_{h,w}})]_t$ followed by the re-encoding with $\mathcal{F}_{\mathsf{ReEncode}}$ is abstracted as a functionality $\mathcal{F}_{\mathsf{Collect}}$, which we implement with a protocol Π_{Collect} in the full version of the paper. The communication complexity of Π_{Collect} is $O(N \cdot n \cdot m + n^2 \cdot m^2 \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$, where S is the set of eliminated parties.

Network Routing for a Circuit Segment. Consider a segment $\mathtt{seg}$. We can finally describe Protocol $\Pi_{\mathsf{NetworkRouting}}(\mathtt{seg})$, which computes sharings $[\phi(\boldsymbol{x})]_t$ for every input group $\boldsymbol{x}$ of the segment $\mathtt{seg}$. In a nutshell, the protocol proceeds exactly as sketched above: (1) $\mathcal{F}_{\mathsf{FanOut}}$ is used to copy wires, and (2) Secret collection is performed by calling $\mathcal{F}_{\mathsf{Permute}} \rightarrow \mathcal{F}_{\mathsf{Collect}} \rightarrow \mathcal{F}_{\mathsf{Permute}}$. However, for our "non-black-box" optimization, suppose that $\Pi_{\mathsf{NetworkRouting}}(\mathtt{seg}')$ was called for the segment $\mathtt{seg}'$ that goes right before $\mathtt{seg}$. In this case, $\mathcal{F}_{\mathsf{FanOut}}$ has been performed on all output groups prior to segment $\mathtt{seg}$, and therefore we only need to do it for these output groups in $\mathtt{seg}$. Even more, since $\Pi_{\mathsf{NetworkRouting}}$ is called to verify the inputs of $\mathtt{seg}$, we only need to use $\mathcal{F}_{\mathsf{FanOut}}$ in these output wires of $\mathtt{seg}$ that are fed as inputs to $\mathtt{seg}$ itself. Later, if the check passes, then we apply $\mathcal{F}_{\mathsf{FanOut}}$ to the remaining wires. A similar optimization happens with the first call to $\mathcal{F}_{\mathsf{Permute}}$, which is done on the output groups.

The description of $\Pi_{\mathsf{NetworkRouting}}$ appears in Protocol 2. If $\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(N)}$ are the output groups of $\mathtt{seg}$, we denote by $\tilde{n}_j^{(i)}$ for $j \in [\ell], i \in [N]$ the number of times that wire $z_j^{(i)}$ is used *inside* $\mathtt{seg}$ itself. Protocol $\Pi_{\mathsf{NetworkRouting}}$ assumes that the protocol has been called for the previous segment, so these are the only remaining copies needed for getting the input groups of $\mathtt{seg}$. After this, the $\mathcal{F}_{\mathsf{Permute}} \rightarrow \mathcal{F}_{\mathsf{Collect}} \rightarrow \mathcal{F}_{\mathsf{Permute}}$ sequence is applied.

Protocol 2: $\Pi_{\mathsf{NetworkRouting}}(\mathtt{seg})$

1. All parties call $\mathcal{F}_{\mathsf{FanOut}}$ on all the output sharings of $\mathtt{seg}$ and $\{\tilde{n}_j^{(i)}\}_{j \in [\ell], i \in [N]}$, where the j-th wire of $\boldsymbol{z}^{(i)}$ is copied $\tilde{n}_j^{(i)}$ times (this is the number of times this wire is used in $\mathtt{seg}$ itself). All parties receive the fan-out sharings which are used for this segment's gate inputs, and a set of eliminated parties denoted by S_1. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_1$.

2. All parties call $\mathcal{F}_{\mathsf{Permute}}$ with the fan-out sharings and the desired permutations as input. All parties receive the permuted fan-out Shamir secret sharings, and a set of eliminated parties denoted by S_2. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_2$.
3. All parties call $\mathcal{F}_{\mathsf{Collect}}$ to get the collected Shamir sharings of `seg`'s gate group inputs. All parties also receive a set of eliminated parties denoted by S_3. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_3$.
4. All parties call $\mathcal{F}_{\mathsf{Permute}}$ with the collected Shamir sharings as input, and get all Shamir sharings of `seg`'s gate group inputs, denoted by $\{[\phi(\tilde{\boldsymbol{x}}^{(i)})]_t\}_{i\in[N]}$ and $\{[\phi(\tilde{\boldsymbol{y}}^{(i)})]_t\}_{i\in[N]}$. All parties also receive a set of eliminated parties denoted by S_4.
5. All parties output $S := S_1 \cup S_2 \cup S_3 \cup S_4$ and all the input Shamir sharings $\{[\phi(\tilde{\boldsymbol{x}}^{(i)})]_t\}_{i\in[N]}$ and $\{[\phi(\tilde{\boldsymbol{y}}^{(i)})]_t\}_{i\in[N]}$.

3.5 Efficient Verification

Now we are ready to "patch" the verification protocol from Sect. 3.3 to get linear communication. The full protocol, Π_{Verify}, is given in the full version of the paper, and here we only discuss the core differences with respect to the protocol from before. As before, we do call $\Pi_{\mathsf{NetworkRouting}}$ in step 5, except that this time we take into account the fact that $\mathcal{F}_{\mathsf{FanOut}}$ and $\mathcal{F}_{\mathsf{Permute}}$ have been called for all previous segments, and hence only need to be computed for the current segment (as described in $\Pi_{\mathsf{NetworkRouting}}$). Second, recall that the $\mathcal{F}_{\mathsf{FanOut}}$ and $\mathcal{F}_{\mathsf{Permute}}$ calls in $\Pi_{\mathsf{NetworkRouting}}$ are only performed to the output groups needed for the input groups in the current segment `seg`. If step 7 succeeds, then the parties need to apply $\mathcal{F}_{\mathsf{FanOut}}$ and $\mathcal{F}_{\mathsf{Permute}}$ to the remaining output groups in `seg` in preparation to the call to $\Pi_{\mathsf{NetworkRouting}}$ for the next segment.

Protocol Π_{Verify} is given in the full version of the paper. In that section we analyze its communication, verifying that it indeed grows linearly with n.

3.6 Fault Localization

Finally, we have focused so far in how the parties detect that cheating occurred, but we have not discussed how to react to that, identifying a semi-corrupted pair so that the segment can be re-run. Recall that a parity element $p \in \mathcal{R}$ is an element that satisfies $\mathsf{val}(p) = 0$. To mask the shares of extended additive sharing $(\!|s|\!)$ when the dealer D opens all shares, all parties will prepare a random degree-t Shamir sharing of a random parity element, denoted by $[p]_t$. Then all parties will send their shares of $(\!|s|\!) + [p]_t$ to D.

We note that it is enough for D to localize a semi-corrupted pair just by opening the two sharings $(\!|s|\!) + [p]_t$ and $\langle s\rangle + \langle o\rangle$, so we introduce the following steps to let all parties disclose to D the randomness masks that they have distributed and received. We first observe that, due to the way all parties prepare random sharings (see the full version of the paper), $\langle o\rangle$ can be written as

$\langle o\rangle = \sum_{i=0}^{n'}\langle o_i\rangle$, and $[p]_t$ can be written as $[p]_t = \sum_{i=0}^{n'}[p_i]_t$, where $\langle o_i\rangle$ is the zero additive sharing distributed by P_i and $[p_i]_t$ is the parity sharing distributed by the party P_i. Each parity sharing $[p_i]_t$ corresponds to an additive sharing whose secret is 0, denoted by $\langle o_i'\rangle$. Let $\langle o'\rangle := \sum_{i=1}^{n'}\langle o_i'\rangle$. Note that D can locally compute $\langle o\rangle - \langle o'\rangle$, and $\langle o\rangle - \langle o'\rangle = \sum_{i=1}^{n'}\langle o_i\rangle - \langle o_i'\rangle$.

Following the idea in [BFO12], in order to protect the shares of $\langle o_i\rangle - \langle o_i'\rangle$ (which may leak information), each party P_i distributes another additive zero sharing $\langle o_i''\rangle$ as a mask. Then P_i reveals to D all the shares of $\langle o_i\rangle - \langle o_i'\rangle + \langle o_i''\rangle$, and also the share of $\langle o_j\rangle - \langle o_j'\rangle + \langle o_j''\rangle$ that P_i received from another party P_j. Given this information, D is able to identify disputes between parties.

We summarize the full protocol $\Pi_{\mathsf{FaultDetection}}$ in Section ?? in the Supplementary Material.

4 Main Protocol

In the previous section we saw how to evaluate each segment in the circuit, and how to check if the execution was correct, identifying a semi-corrupt pair if this was not the case. In this section we show how to put together these protocols in order to evaluate the entire circuit with G.O.D., essentially by making use of the player elimination framework by Beerliová-Trubíniová and Hirt [BTH08], in which the parties that remain after a semi-corrupted pair is removed re-run the failed segment. This is described in Sect. 4.3. No MPC protocol would be complete without describing how input and output gates are handled. This is explained in Sects. 4.1 and 4.2, respectively.

4.1 Input Gates

Since we are in the client-server model, all the inputs belong to the clients. Recall that we assume that the number of inputs for each client is a multiple of ℓ. In this part, we introduce a protocol Π_{Input}, which enables all client to share their inputs to all parties, and then properly performs fan-out and permutations to input sharings to prepare them for later use. We describe the protocol Π_{Input} below.

Protocol 3: Π_{Input}

1. For each client Client, suppose its inputs are $\{x_i^{(1)},\ldots,x_i^{(\ell)}\}_{i=1}^N$. All parties and Client perform the following steps:
   (a) Let $\boldsymbol{x}_i := (x_i^{(1)},\ldots,x_i^{(\ell)}) \in (\mathbb{Z}/2^k\mathbb{Z})^\ell$. Client locally computes $\phi(\boldsymbol{x}_i)$ for all $i \in [N]$.
   (b) All parties call $\mathcal{F}_{\mathsf{InputShamir}}$ with the inputs $\phi(\boldsymbol{x}_1),\ldots,\phi(\boldsymbol{x}_N)$ from Client. All parties get the output $[y_1]_t,\ldots,[y_N]_t$ and a set of eliminated parties denoted by S_1. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_1$.
   (c) All parties call $\mathcal{F}_{\mathsf{RandShamir}}(N)$ to prepare N random degree-t Shamir secret sharings denoted by $[\phi(\boldsymbol{r}_1)]_t,\ldots,[\phi(\boldsymbol{r}_N)]_t$. All parties also

receive a set of eliminated parties denoted by S_2. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_2$.

(d) All parties call $\mathcal{F}_{\mathsf{OpenPub}}(N)$ to reconstruct $[y_1]_t + [\phi(\boldsymbol{r}_1)]_t, \ldots, [y_N]_t + [\phi(\boldsymbol{r}_N)]_t$ and get the secrets $y_1 + \phi(\boldsymbol{r}_1), \ldots, y_N + \phi(\boldsymbol{r}_N)$.

(e) For all $i \in [N]$, All parties locally check if $\phi \circ \psi(y_i + \phi(\boldsymbol{r}_i)) = y_i + \phi(\boldsymbol{r}_i)$. If the equation does not hold, all parties set $[y_i]_t$ to $[0]_0$. Note that after this step, each sharing $[y_i]_t$ can be written as $[y_i]_t = [\phi(\boldsymbol{x}'_i)]_t$, where $\boldsymbol{x}'_i = \psi(y_i)$.

(f) All parties take $[\phi(\boldsymbol{x}'_1)]_t, \ldots, [\phi(\boldsymbol{x}'_N)]_t$ as the shared inputs of Client.

2. All parties call $\mathcal{F}_{\mathsf{FanOut}}$ with the input sharings from all clients, and get the set of eliminated parties denoted by S_3 and the resulting fan-out sharings. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_3$.
3. All parties call $\mathcal{F}_{\mathsf{Permute}}$ to permute all the fan-out sharings with the desired permutations. All parties get the set of eliminated parties denoted by S_4, and the resulting permuted sharings. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_4$.
4. Let S be the set of eliminated parties in the previous steps. All parties output the resulting permuted sharings in the previous step.

Cost of Π_{Input}. To get input from a client with $N \cdot \ell$ inputs in $\mathbb{Z}/2^k\mathbb{Z}$, the communication complexity is $O(N \cdot n \cdot m + n^2 \cdot m^2 \cdot (|S_1| + |S_2|))$ elements in $\mathbb{Z}/2^k\mathbb{Z}$. The communication complexity of Step 2 is $O(M \cdot n \cdot m + n^2 \cdot m^2 \cdot |S_3|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$, and the communication complexity of Step 3 is $O(M \cdot n \cdot m + n^2 \cdot m^2 \cdot |S_4|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$. Since M is bounded by $O(|C'|/\ell)$, and $m, \ell = O(\log n)$, the total communication complexity of Π_{Input} is $O(|C| \cdot n + c \cdot n \cdot m + n^2 \cdot m^2 \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$.

4.2 Output Gates

We introduce the protocol Π_{Output} that reconstructs all outputs towards all clients. In this protocol, all parties first perform network routing to generate the output Shamir sharings, and then they send the shares of the output sharing to each client. The description of Π_{Output} appears below.

Protocol 4: Π_{Output}

1. All parties call $\mathcal{F}_{\mathsf{Collect}}$ to collect secrets for Shamir sharings of the output layer, and all parties receive a set of eliminated parties denoted by S_1. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_1$.
2. All parties call $\mathcal{F}_{\mathsf{Permute}}$ with the collected Shamir sharings and the desired permutations as input. All parties get all output Shamir sharings as output, and get a set of eliminated parties denoted by S_2. All parties update $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} - S_2$.
3. For each client Client that has output Shamir sharings $[\phi(\boldsymbol{z}_1)]_t, \ldots, [\phi(\boldsymbol{z}_N)]_t$, all parties send their shares of $[\phi(\boldsymbol{z}_i)]_t$ to Client for all $i \in [N]$. Then Client reconstructs the secrets $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N$.

Cost of Π_{Output}. In Step 1, the communication complexity is bounded by calling $\mathcal{F}_{\mathsf{Collect}}$ that outputs $|C'|/\ell$ sharings, so the communication complexity is bounded by $O((|C| + \ell \cdot c + n^2 \cdot m^2) \cdot n \cdot m/\ell + n^2 \cdot m^2 \cdot |S_1|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$. In step 2, the communication complexity is bounded by $O((|C| \cdot n + \ell \cdot c + n^2 \cdot m^2) \cdot n \cdot m/\ell + n^2 \cdot m^2 \cdot |S_2|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$. Since m, ℓ are both $O(\log n)$, the total communication complexity of Π_{Output} is $O(|C| \cdot n + c \cdot n \cdot m + n^3 \cdot m^2)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$.

4.3 Main Protocol

Given the above protocols, the main protocol that implements the ideal functionality $\mathcal{F}_{\mathsf{Main}}$ follows in a straightforward way. The main protocol is introduced in Π_{Main}.

Protocol 5: $\Pi_{\mathsf{Main}}(C)$

1. Let C denote the circuit. All parties transform C to C'. All parties agree on the gate grouping, and they order of the circuit segments according to topological ordering. All parties set $\mathcal{P}_{\mathtt{active}} := \mathcal{P}$.
2. All parties run the protocol Π_{Input}.
3. All parties evaluate the circuit segments according to their ordering. For each circuit segment denoted by `seg`:
   (a) All parties run the protocol $\Pi_{\mathsf{Eval}}(\mathtt{seg})$. All parties get the set of eliminated parties denoted by S.
   (b) All parties run the protocol $\Pi_{\mathsf{Verify}}(\mathtt{seg}, S)$. If the output is `incorrect`, all parties repeat step 3.(a) and 3.(b) to evaluate `seg`. Otherwise, all parties continue to evaluate the next circuit segment.
4. All parties run the protocol Π_{Output}.

Cost of Π_{Main}. In Step 2, the cost of Π_{Input} is In Step 3, each time Π_{Eval} or Π_{Verify} is repeated, the communication complexity is $O(|C| + n^2 \cdot m^2 \cdot |S|)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$.

In Step 3, when no circuit segment is repeated, the total communication complexity of Π_{Eval} and Π_{Verify} is $\sum_{\mathtt{seg}} \big(O(n^2 \cdot m^2 \cdot |S|) + O(|C|) + O(M_{\mathtt{seg}} \cdot n \cdot m)\big)$. We note that $\sum_{\mathtt{seg}} M_{\mathtt{seg}}$ is bounded by $O(|C'|/\ell)$, and that $\sum_{\mathtt{seg}} |S|$ is bounded by $2t = O(n)$. So the total communication complexity of Step 3 when no circuit segment is repeated is $O((|C'|/\ell) \cdot n \cdot m + |C| \cdot n + n^3 \cdot m^2)$.

Recall that $m, \ell = O(\log n)$. Also note that the sum of all $|S|$ is bounded by $2t = O(n)$. Therefore, the overall communication complexity of Π_{Main} is $O(|C| \cdot n + c \cdot n \cdot \log n + n^3 \cdot \log^2 n)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$. Here $|C|$ is the size of the circuit in *both* addition and multiplication gates. Notice that this is *linear*, as desired.

Lemma 3. *Protocol Π_{Main} securely computes $\mathcal{F}_{\mathsf{Main}}$ in the ($\mathcal{F}_{\mathsf{InputShamir}}$, $\mathcal{F}_{\mathsf{RandShamir}}$, $\mathcal{F}_{\mathsf{RandZeroAdd}}$, $\mathcal{F}_{\mathsf{RandParity}}$, $\mathcal{F}_{\mathsf{OpenPub}}$, $\mathcal{F}_{\mathsf{ReEncode}}$, $\mathcal{F}_{\mathsf{Mult}}$, $\mathcal{F}_{\mathsf{TripleAdd}}$,*

$\mathcal{F}_{\mathsf{TripleMult}}$, $\mathcal{F}_{\mathsf{VerifyBC}}$, $\mathcal{F}_{\mathsf{FanOut}}$, $\mathcal{F}_{\mathsf{Permute}}$, $\mathcal{F}_{\mathsf{Collect}}$)-hybrid model with perfect security against a fully malicious adversary who controls $t < n/3$ parties.

The proof of Lemma 3 can be found in the full version of the paper. This leads to the following Theorem, which is the main result of our work.

Theorem 3. *In the client-server model, let c denote the number of clients, and $n = 3t + 1$ denote the number of parties (servers). Let k be a constant positive integer and let $\mathbb{Z}/2^k\mathbb{Z}$ be a finite ring of constant size. For an arithmetic circuit C over $\mathbb{Z}/2^k\mathbb{Z}$, let $|C|$ denote the size of the circuit. There exists an information-theoretic MPC protocol which securely computes the arithmetic circuit with perfect security in the presence of a fully malicious adversary controlling up to c clients and t parties. The communication complexity of this protocol is $O(|C| \cdot n + c \cdot n \cdot \log n + n^3 \cdot \log^2 n)$ elements in $\mathbb{Z}/2^k\mathbb{Z}$.*

Acknowledgments. This paper was prepared in part for information purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful. 2024 JP Morgan Chase & Co. All rights reserved.

Y. Song was supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61361136003.

References

[ACD+19] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019: 17th Theory of Cryptography Conference, Part I*, volume 11891 of *Lecture Notes in Computer Science*, pages 471–501, Nuremberg, Germany, December 1–5, 2019. Springer, Heidelberg, Germany.

[ACE+21] Mark Abspoel, Ronald Cramer, Daniel Escudero, Ivan Damgård, and Chaoping Xing.Improved single-round secure multiplication using regenerating codes.In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part II*, volume 13091 of *Lecture Notes in Computer Science*, pages 222–244, Singapore, December 6–10, 2021. Springer, Heidelberg, Germany.

[BFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority.In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 663–680, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 1–10, New York, NY, USA, 1988. Association for Computing Machinery.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008: 5th Theory of Cryptography Conference*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230, San Francisco, CA, USA, March 19–21, 2008. Springer, Heidelberg, Germany.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols.*Journal of Cryptology*, 13(1):143–202, January 2000.

[CCXY18] Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized complexity of information-theoretically secure MPC revisited.In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 395–426, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[CRX21] Ronald Cramer, Matthieu Rambaud, and Chaoping Xing. Asymptotically-good arithmetic secret sharing over $\mathbb{Z}/p^\ell\mathbb{Z}$ with strong multiplication and its applications to efficient MPC. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part III*, volume 12827 of *Lecture Notes in Computer Science*, pages 656–686, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.

[DEF+19] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography.In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 445–465, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany.

[ELXY23] Daniel Escudero, Hongqing Liu, Chaoping Xing, and Chen Yuan. Degree-d reverse multiplication-friendly embeddings: Constructions and applications. *Asiacrypt*, 2023.

[FR22] Thibauld Feneuil and Matthieu Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. Cryptology ePrint Archive, Report 2022/1407, 2022. https://eprint.iacr.org/2022/1407.

[GLS19] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery.In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 85–

114, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.

[GPS21] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 275–304, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.

[GPS22] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority MPC with packed secret sharing. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 3–32, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany.

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 618–646, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[HMP00] Martin Hirt, Ueli M. Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In Tatsuaki Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 143–161, Kyoto, Japan, December 3–7, 2000. Springer, Heidelberg, Germany.

[IKP+16] Yuval Ishai, Eyal Kushilevitz, Manoj Prabhakaran, Amit Sahai, and Ching-Hua Yu. Secure protocol transformations.In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 430–458, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany.

[PS21] Antigoni Polychroniadou and Yifan Song. Constant-overhead unconditionally secure multiparty computation over binary fields. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 812–841, Zagreb, Croatia, October 17–21, 2021. Springer, Heidelberg, Germany.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[Wan03] Zhe-Xian Wan. *Lectures on finite fields and Galois rings.* World Scientific Publishing Company, 2003.

Compute, but Verify: Efficient Multiparty Computation over Authenticated Inputs

Moumita Dutta[1](✉), Chaya Ganesh[1], Sikhar Patranabis[2], and Nitin Singh[2]

[1] Indian Institute of Science, Bangalore, India
{moumitadutta,chaya}@iisc.ac.in
[2] IBM Research, Bangalore, India
sikhar.patranabis@ibm.com, nitisin1@in.ibm.com

Abstract. Traditional notions of secure multiparty computation (MPC) allow mutually distrusting parties to jointly compute a function over their private inputs, but typically do not specify how these inputs are chosen. Motivated by real-world applications where corrupt inputs could adversely impact privacy and operational legitimacy, we consider a notion of *authenticated* MPC where the inputs are authenticated (for instance, signed using a digital signature) by some certification authority. We propose a generic and efficient compiler that transforms any linear secret sharing based honest-majority MPC protocol into one with input authentication.

Our compiler achieves an ideal notion of authenticated MPC equipped with stronger and more desirable security guarantees than those considered in prior works, while incurring significantly lower computational costs and competitive communication overheads when compared to existing solutions. In particular, we entirely avoid the (potentially expensive) protocol-specific techniques and pre-processing requirements that are inherent to these solutions. For certain corruption thresholds, our compiler additionally preserves the stronger identifiable abort security of the underlying MPC protocol. No existing solution for authenticated MPC achieves this regardless of the corruption threshold.

Along the way, we make several technical contributions that are of independent interest. This includes the notion of distributed proofs of knowledge and concrete realizations of the same for several relations of interest, such as proving knowledge of many popularly used digital signature schemes, and proving knowledge of opening of a Pedersen commitment.

1 Introduction

Secure multiparty computation (MPC) [7,32,38,51,52] allows two or more parties to jointly compute a function of their private inputs, while ensuring input privacy and output correctness (even in the presence of some corrupt parties). Traditional security notions for MPC ensure *output correctness* and *input privacy*, that is, nothing is leaked about the parties' private inputs beyond the (correct) output of the computation. However, no assurance is given about how the parties choose their private inputs.

Unfortunately, certain applications of MPC could be sensitive to "ill-formed inputs". Maliciously chosen inputs could either corrupt the output or reveal the output on arbitrary inputs, thus violating the desired real-world security guarantees of an MPC protocol. Such attacks are not captured by traditional MPC security definitions.

Input Authenticity in MPC. There are several real-world applications of MPC where it is important to ensure that the inputs used by parties are *authentic*. If a set of individuals on a job portal wish to compute "industry average compensation" for their expertise and experience in a privacy preserving manner (e.g., services provided by glassdoor), one would want them to input payslips bearing their employers' signature. Similarly, in applications involving hospitals performing joint computations on patient data for treatment efficacy, it is desirable to ensure that the data used is signed by a regulatory authority. Input validation is also of practical relevance in applications of MPC in computation on genomic data [10]. For all of these applications, the traditional MPC security guarantees are clearly inadequate. A natural question that confronts us then is: *how do we ensure that authentic inputs are used in MPC*?

Authentication via Certification. In the real world, data authentication typically involves the data being attested by a relevant *certifying authority*. In our work, we specifically consider applications where an input bearing a signature is considered *authentic* and we can assume the existence of a relevant certifying authority that provides the signature. For instance, employers can act as the certification authority to digitally sign the payslips when parties wish to compute 'industry average compensation' using services like glassdoor, a financial auditor can act as the certification authority to digitally sign the bills of sale when shipping companies wish to compute aggregate statistics on private data, a regulatory authority (like WHO) act as the certification authority to digitally sign the medical records when hospitals wish to perform joint computation over sensitive patient data, and so on. Since the certifying authority cannot be omnipresent to vouch for authenticity of the data, it is increasingly common for individuals to claim this attestation through *digital signatures* that can be verified efficiently. In fact, there exist several digital signature schemes today [12,17,44] that allow establishing attestation by a certifying authority while requiring minimal disclosure of attributes, and while maintaining *unlinkability* (several usages of the same credential cannot be linked to the same individual). Unfortunately, such secure mechanisms for authenticating data in the individual context do not translate when computing over data from *multiple* data owners using vanilla MPC protocols (that do not consider input authentication).

Potential Approaches and Pitfalls. A naïve approach would be to incorporate input authentication *as part of the function* to be computed. However, this is practically inefficient. For example, incorporating signature verification as part of the function would entail performing expensive operations such as hashing inside MPC (typically, most signature schemes hash the message), and would also require expressing the algebraic operations underlying signature verification as arithmetic circuits. This significantly blows up the size of the circuit, rendering the resulting MPC protocol practically inefficient.

A more efficient alternative is to have the certifying authority *sign a commitment* (e.g., a Pedersen commitment [41]) to each input, and then have the parties prove that their inputs are those contained inside the public commitments (using customized zero-knowledge proofs). However, this fails to provide *unlinkability*, which is an essential privacy requirement. In particular, one can use the signed commitment to link different protocols where the same input is reused. The alternative would be to get the certifying authority to sign a different commitment for each protocol execution, which again requires the authority to be omnipresent, and is clearly impractical.

Certain prior works [2,11] proposed using *authenticated secret-sharing* in order to certify inputs to an MPC protocol. However, authenticated secret-sharing only provides stand-alone guarantees about the shares themselves, and additional techniques would be needed to ensure that malicious parties actually use these authenticated shares in the execution of the actual MPC protocol (the details of such techniques are not specified completely in prior works [2,11]). Ideally, we want a notion that *ties* input authentication with the underlying MPC, thus preventing malicious parties from using inputs different from the authenticated ones.

Our Goal. We aim to *lift* existing MPC protocols into authenticated ones that ensure that an additional predicate is satisfied by each input (for instance, each input is signed by a common certifying authority). We want to achieve such input authentication (i) without changing the underlying MPC protocol, (ii) without representing the predicate as a circuit, (iii) incurring communication overhead that is succinct in the size of the inputs (which are typically large for the applications we consider), and (iv) maintaining unlinkability. These requirements immediately preclude prior approaches requiring the authentication relation to be expressed as a circuit [13,34], as well as the natural approach based on signed public commitments outlined above, which lacks unlinkability.

1.1 Our Contributions

In this work, we study *authenticated MPC*. We present the first *generic compiler* than efficiently augments existing MPC protocols to additionally ensure that each input has a valid attestation (in the form of a digital signature) from a relevant certifying authority, while retaining both practical efficiency and unlinkability. We illustrate the compatibility of our proposed approach with popularly used privacy-preserving verifiable attestation mechanisms based on digital signatures such as BBS+ [4,12] and PS [44]. Towards this goal, we put forth a notion of distributed (zero-knowledge) proof of knowledge that is of independent interest.

Distributed Proof of Knowledge (DPoK). In Sect. 3, we put forth a notion of a *distributed proof of knowledge* (abbreviated as (DPoK)). A DPoK works in a setting with multiple provers and a single verifier, where the witness is secret shared among the provers. Concretely, for a relation $\mathcal{R}$ and an instance-witness pair $(x, w) \in \mathcal{R}$, the verifier holds the (public) instance x, and each prover holds a share w_i of the (secret) witness w such that $w = \mathsf{Reconstruct}(w_1, \ldots, w_n)$. We also assume a restricted communication model: (i) the provers do not communicate with each other, and (ii) the verifier communicates only via a broadcast channel and is public coin (this facilitates *public verifiability*, which is used crucially in our eventual solution for authenticated MPC). Our definition of DPoK may thus be viewed a natural distributed analogue of honest-verifier public coin protocols.

Robust Complete DPoK. Our basic DPoK definition does not prevent malicious provers from disrupting protocol execution, and only provides *security with abort*. To tackle this, we introduce a stronger notion of *robust completeness* for a DPoK, which additionally provides tolerance against abort in the presence of (a potentially smaller number of) maliciously corrupt provers. Looking ahead, using robust complete DPoKs allows us to achieve authenticated MPC protocols with stronger security guarantees.

DPoK for Discrete Log. In Sect. 3, we also construct a DPoK for the discrete logarithm relation, where the witness (the discrete log of a publicly known group element) is secret-shared (using Shamir secret sharing) across multiple provers. Notably, our construction achieves: (i) *succinct* communication (logarithmic in the size of the witness), and (ii) robust completeness (which ensures that the protocol accepts even in the presence of up to $n/3$ malicious provers, where provers only holds shares to the correct witness). For succinct communication, we use techniques due to Attema et al. [3] to *compress* the communication complexity of our protocol from linear to logarithmic in the size of the witness. We realize robust completeness via error-correction *in the exponents* of group elements. To this end, we leverage results from low degree testing used in prior works to construct efficient zkSNARKs (such as in [1,8]). While achieving robust completeness is straightforward if we do not care about succinctness (and vice versa), the main technical novelty of our construction is to achieve *both* properties simultaneously.

In Appendix C of the full version of the paper [24], we present a generalization of the above DPoK for discrete log that works with *any threshold linear secret sharing scheme*. In this generalized version, we characterize the corruption threshold for robust completeness in terms of the minimum distance of the linear code associated with the threshold linear secret sharing scheme. As an example, we derive concrete bounds on the corruption threshold for the popularly used *replicated secret sharing* scheme.

DPoKs for Algebraically Structured Signatures. Our DPoK for discrete log can be used to build a DPoK for any digital signature scheme where the associated proof of knowledge of a signature can be modeled as a proof of knowledge of the opening of a Pedersen commitment. We present specific instances of this general approach for signature schemes that are algebraically compatible, namely

BBS+ [4,12,15][1] (detailed in Sect. 4) and PS [44] (detailed in Appendix G of the full version of the paper [24]). These signature schemes are popular candidates for applications such as verifiable credentials for self-sovereign digital identity. While these signature schemes natively support efficient (albeit non-distributed) zero-knowledge proofs of knowledge of a valid message-signature pair, our work introduces the first practically efficient DPoKs for these signature schemes that are both succinct and robust complete. Our techniques are modular, and we believe that they can be extended to yield DPoKs for other algebraically structured signatures such as [16], as well as algebraic relations of interest for other applications.

Round Efficient DPoKs in the ROM. The above definitions and constructions of DPoKs are in the standard model. In Appendix E of the full version of the paper [24], we formally define *round efficient* DPoKs in the random oracle model (ROM). This definition is based on the Fiat-Shamir heuristic [25], using which we transform a DPoK (with number of rounds logarithmic in the size of the witness) into a round efficient DPoK (having constant number of rounds). Under this definition, we present round efficient versions of our DPoK constructions for discrete log and algebraically structured signatures; these protocols achieve the same robust completeness and succinct communication guarantees as the original protocols, albeit in the ROM.

Authenticated MPC. We now expand upon our main contribution, namely *authenticated MPC.* Informally, we consider a notion of input authenticity for MPC where each input is certified using a valid signature from a certification authority. This is standard in applications where a publicly known certifying authority (external to the MPC protocol) signs an input to certify that the input satisfies certain properties[2]. We build upon our DPoKs for BBS+ and PS signatures to propose a generic compiler that transforms any (threshold linear) secret-sharing based maliciously secure honest-majority MPC protocol into its *authenticated* MPC version. Our compiler yields the first practically efficient MPC protocols that satisfy an ideal notion of input authenticity while preserving practical efficiency and unlinkability. We prototype-implement a specific instance of our compiler that achieves input authentication based on our proposed DPoK for BBS+ signatures. Finally, we present experimental results to illustrate that our compiler incurs *negligible communication overhead* over the original MPC protocol. For simplicity, our ideal functionality and subsequent protocols are described assuming a common signature authority for all inputs. The more general case involving multiple signing authorities also follows with minor modifications without incurring any loss of efficiency.

Ideal Functionality for Authenticated MPC. In Sect. 5, we formalize the above notion for authenticated MPC via an ideal functionality $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{auth}}$ that works as

[1] There are standardization efforts for using BBS+ signatures in verifiable credentials for Web 3.0, leading to a recent RFC draft [39].

[2] Our techniques extend to other notions of authenticity such as proving that the inputs open publicly known commitments.

follows. The parties send their inputs x_i and signature σ_i on x_i to $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{auth}}$ for $i \in [n]$. The functionality $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{auth}}$ then checks if σ_i is a valid signature on x_i for all $i \in [n]$. For each $j \in [n]$ such that σ_j is not a valid signature on x_j, $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{auth}}$ sends $(\texttt{abort}, P_j)$ to all of the parties. Otherwise it computes $y = f(x_1, \ldots, x_n)$ and outputs y to all of the parties.

We note that our ideal functionality ties input authentication into the underlying MPC, thus preventing malicious parties from using different inputs as compared to the authenticated ones. The prior works [2,11] only provide stand-alone guarantees about the authenticated shares themselves, and requires additional techniques to ensure that these authenticated shares are then used in the execution of the actual MPC protocol. We further note that our ideal functionality already captures unlinkability, since the adversary does not learn any additional information about the authenticated input (beyond the function output) that might allow it to correlate the usage of the same input-signature pair across multiple executions. This rules out solutions based on signing public commitments to inputs, which trivially violate unlinkability.

Compiler for Authenticated MPC. In Sect. 5, we present a compiler that transforms any Shamir secret-sharing based maliciously secure honest-majority MPC protocol Π into its *authenticated* MPC version Π' that securely realizes the above ideal functionality $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{auth}}$, where each input is authenticated using a BBS+ signature. Our compiler builds upon our DPoK for BBS+ signatures from Sect. 4. In Appendix G of the full version of the paper [24], we present an analogous compiler for input authentication using PS signatures, which builds upon our DPoK for PS signatures. In both cases, the compiled protocol Π' inherits the security of Π as long as the inputs are authentic (by definition, we abort if this is not the case)[3]. If Π guarantees security with identifiable abort, then the same holds for Π'. If Π achieves guaranteed output delivery, then so does Π' (albeit for a corruption threshold $t < n/3$) – this crucially uses the *robust completeness* property of the underlying DPoKs.

Generalization and Extensions. We note that our approach works in general for: (a) any (threshold linear) secret-sharing based MPC protocol, and (b) any signature scheme such that the associated proof of knowledge can be modeled as a proof of knowledge of the opening of a Pedersen commitment (such as CL signatures [16] and PS signatures [44]). Our DPoK-based approach also offers the flexibility of extending our compiler to support other notions of input authentication, beyond proving knowledge of signatures. In particular, one can build upon our approach to prove a wider class of expressive predicates over secret-shared inputs, thus catering to a wide range of applications with diverse proof requirements (e.g., federated learning). For instance, each party can publish a commitment to its input at the beginning of the authenticated MPC protocol, and then use our DPoK-based framework to prove the following simultaneously:

[3] In some applications, it is acceptable to continue computation on default inputs instead of aborting when authentication fails.

(i) the secret-shared input is signed by a certifying authority (this follows from the basic compiler), (ii) the secret-shared input is a valid opening to the published commitment, and (iii) the opening to the commitment satisfies a certain predicate. Note that, if a different application requires new/additional properties to be checked, the aforementioned approach avoids the need to involve the certifying authority each time. Similarly, it maintains unlinkability since a fresh commitment is used for each protocol execution, while the DPoK allows keeping the signature from the certifying authority private.

Implementation and Experiments. In Sect. 6, we present a prototype implementation of our BBS+ based authenticated MPC protocol, and illustrate that our approach incurs very little computational and communication overheads over and above the original MPC protocol. In particular, we implement the BBS+ based instance of our compiler and use it to transform an implementation of a native MPC (instantiated via MP-SPDZ [22,36,37]) into an authenticated MPC. We use this implementation to benchmark an application of authenticated MPC, where n shipping companies with private datasets wish to securely compute aggregate statistics on some subset of their combined data. Note that this is an application where the number of inputs of each party is much larger than the number of parties involved in the protocol. Specifically, we consider each dataset $D_i = (C_i, S_i)$ to be partitioned into k *categorical* columns C_i and ℓ numeric columns D_i. A sample query specifies $\{(j, v_j)\}_{j \in J}$ for $J \subset [k]$. The goal is to compute means of numeric columns on the subset of rows satisfying the selection predicate $C[j] = v_j$ for $j \in J$, i.e. the subset of rows with specified values of some categorical features. We also assume an external certifying entity $\mathcal{T}$ (e.g. a financial auditor) which independently verifies the correctness of sales data reported by different organizations and issues a digital signature to attest the same (this entity does not participate in the MPC protocol).

We conduct experiments to evaluate the computational and communication overheads incurred by our protocol to achieve authentication on top of native MPC. The results are summarized in Table 1. For comparison, we also show the: (i) the actual computational and communication overheads for the native MPC protocol, and (ii) the computational and communication overheads incurred by an alternative approach of authenticating the inputs that shows the consistency of the input shares with a public digest of the input and proves knowledge of a BBS+ signature on this public digest by expressing the verification algorithm as an arithmetic circuit and evaluating it inside the MPC protocol. As demonstrated by the results in Table 1, the overheads for this alternative approach are substantial even when an MPC-friendly hash function like MiMC is used to hash the input. In comparison, the overheads for our DPoK-based approach are significantly smaller, and effectively minor when compared to the overheads for the base MPC protocol.

1.2 Technical Overview

In this section, we provide a brief overview of our techniques. We begin by outlining ideas to distribute a well-known protocol for proving knowledge of

Table 1. Comparison of our DPoK-based approach for MPC input authentication with the naïve approach of validating BBS+ signatures inside MPC (which involves computing MiMC hashes inside MPC). These results correspond to datasets of size 500 × 10 in the KPI application.

# Parties	Vanilla MPC	Auth MPC with MiMC Hash	DPoK Overhead
3	33s/8437 MB	273s/13979 MB	5.7s/14.4 KB
5	125s/43823 MB	1369s/14498 MB	6.2s/30 KB
7	386.2s/127057 MB	3645.33s/207427 MB	8.2s/52 KB

discrete logarithm of a public group element. This relation will be at the core of expressive algebraic relations that we will consider later.

Proof of Knowledge of Discrete Log. Let $\mathbb{G}$ be a group of prime order p. Given $x \in \mathbb{G}$, recall Schnorr's protocol [45,46] for proving knowledge of discrete logarithm w such that $x = g^w$ for some generator g (here (g, x) is public and w is the secret witness). Let $(\mathcal{P}^1, \mathcal{P}^2, \mathcal{V})$ be the protocol where we denote by $\mathcal{P}^1$ and $\mathcal{P}^2$ the algorithms that compute, the prover's first message $a = g^\alpha$ for random $\alpha \in \mathbb{F}_p$, and the prover's last message (response) $z = \alpha + cw$, respectively, where c is the challenge from the space $\{0, 1\}^l$ for some length l. Let $\mathcal{V}$ be the algorithm that takes x, transcript $\tau = (a, c, z)$ and accepts iff $g^z = ax^c$.

DPoK for Discrete Log. In order to *distribute* the above protocol, we begin by assuming n provers $\mathcal{P}_i$ who each hold a share w_i such that $w = w_1 + \cdots + w_n \pmod p$. Now, each prover runs Σ with their respective shares in parallel[4]. That is, $\mathcal{P}_i$ runs $\mathcal{P}^1$, broadcasts $a_i = g^{\alpha_i}$, receives challenge c from $\mathcal{V}$, and runs $\mathcal{P}^2$ and broadcasts z_i. The transcript is $\tau = (a_1, \ldots, a_n, c, z_1, \ldots, z_n)$, and the verifier accepts iff $g^{\Sigma z_i} = \prod a_i x^c = \prod_i a_i x^c$. This holds since $g^{\Sigma z_i} = g^{\Sigma(\alpha_i + cw_i)} = \prod_i a_i x^c$.

This idea generalizes to any linear secret sharing scheme, and also extends to other relations. For instance, to prove knowledge of representation of a vector of discrete logarithms with respect to public generators. In our final construction we use additional ideas like randomization of the first message of each $\mathcal{P}_i$ via a sharing of 0 in order to ensure zero-knowledge. This DPoK has communication

[4] This is a simplified description; in our actual protocol Π_{dlog} (Sect. 3.2), there are no parallel sessions, each instance uses a random share, ensuring that we do not reuse the shares, and in the FS-compiled version $\Pi_{\mathsf{dlog}}^{\mathsf{FS}}$ (Appendix E of the full version of the paper [24]), parties send non-interactive proofs instead of sending the first-messages separately in parallel. We note that ROS attacks [9] in the context of concurrent signatures are therefore inapplicable in our setting. See also Sect. 1.4 for a more detailed discussion.

complexity linear in the size of the witness. To achieve succinctness, we instead use as a starting point a compressed sigma protocol [3] in order to achieve a distributed protocol with logarithmic communication complexity (see Sect. 3.2 for details).

Robust Completeness. While the ideas described above result in protocols that are zero-knowledge and sound against a malicious adversary controlling up to t parties, completeness is guaranteed only if all the provers follow the protocol. However, in the distributed setting, a stronger, but natural notion is a *robust* completeness property where completeness holds as long as the shares reconstruct a valid witness, even if some provers are malicious. The main technical challenge in achieving robust completeness for a distributed proof is to retain succinctness. Our key technical novelty is to achieve both robustness and succinctness *simultaneously* via ideas from low-degree testing. We achieve this by identifying and discarding corrupt shares. At a high level, the provers commit to their shares and then reveal a certain linear form determined by the challenge over their shares. Given a challenge $\boldsymbol{c} \in \mathbb{F}_p^m$, each $\mathcal{P}_i$ broadcasts $z_i = \langle \boldsymbol{c}, \mathbf{w}_i \rangle$. In the honest case, these opened linear forms are expected to be a sharing of the same linear form on the reconstructed witness: $\mathbf{z} = (z_1, \ldots, z_n)$ recombine to z where $z = \langle \boldsymbol{c}, \mathbf{w} \rangle$. The verifier error-corrects the received $\mathbf{z}'$ to the nearest codeword, and identifies the erroneous positions. By assumption our corruption threshold is smaller than half the minimum distance of the code, so the erroneous positions clearly come from corrupt provers. Can some corrupt provers strategically introduce errors in individual shares so that they "cancel out" in the inner product with $\mathbf{c}$? We lean on coding theoretic result (Lemma 2 of the full version of the paper [24]) for linear codes to claim that such a prover only succeeds with negligible probability. Finally, having identified the corrupt messages, we can reconstruct the claimed commitment in the exponent using commitments of honest shares (now identified). We need more details around this core idea to ensure the protocol is zero-knowledge (see Sect. 3.2 for a complete treatment).

DPoKs for Algebraically Structured Signatures. It turns out that the above approach can be naturally generalized to obtain a DPoK for the opening of a Pedersen commitment [42]. We use this observation as a starting point to realize DPoKs for algebraically structured signatures such as BBS+ [4,12,15] and PS [44], which naturally admit proofs of knowledge that can be cast as proving knowledge of openings of Pedersen commitments. As a core technical contribution, we introduce a modified proof of knowledge for the BBS+ signature scheme, which leads to a vastly more efficient DPoK as compared to the straightforward approach of distributing prior proofs of knowledge for BBS+ signatures. We refer to Sect. 4 for details. Analogous DPoK for PS signatures is presented in Appendix G of the full version of the paper [24].

Compiler for Authenticated MPC. In order to construct an authenticated MPC protocol, we build upon the above DPoKs for BBS+ and PS signatures. Our compiler reuses the input sharing that is already done as part of an honest-majority MPC protocol. Before proceeding with computation on the shares, the distributed zero-knowledge proof is invoked to verify authenticity, and then the rest of the MPC protocol proceeds. Since the shares of the witness come from a party in the MPC protocol, our robustness property guarantees that if the dealer is honest (that is, a valid witness was shared), then even if some parties acting as provers are dishonest, the authenticity proof goes through (see Sect. 5 for details).

We also note that, while we rely on broadcast for our protocols, all relevant related work on FLPCP [13] and previous works on authenticated MPC [2,11,34] also make use of a broadcast channel. A broadcast channel is not a limitation, and can be implemented using point-to-point channels. In the setting where the number of parties is not too large (as in the applications we consider), the communication overhead to realize broadcast is not prohibitive.

1.3 Related Work

We summarize some relevant related work, and compare our compiler with prior approaches for authenticated MPC. We refer to Appendix A of the full version of the paper [24] for some additional discussions.

Certified Inputs. The earlier works of [5,35,53] achieve input validation for the special case of *two*-party computation using garbled circuit (GC) based techniques. Another work [11] constructs MPC with certified inputs, albeit using techniques that are specific to certain MPC protocols [20,21]. A recent work [2] develops techniques for computing bilinear pairings over secret shared data, which aims to enable signature verification inside MPC for the PS signature scheme [44]. Both works [2,11] emulate a functionality similar to authenticated secret-sharing protocol, where shares of an input certified by some certification authority are provided at the end of the protocol execution. While the goal of authenticated MPC has been studied, these works would require additional consistency checks to ensure the consistency of shares used across the protocols for authentication of shares and the underlying MPC execution. Although the explicit details are not provided in the protocol description, we expect the requirement of some consistency check on the MACs to ensure the usage of same shares during authentication protocol and original MPC for function computation. In our work, we formalize this notion of authenticated MPC as an ideal functionality which incorporates the consistency checks, and prove that the proposed constructions realize this. For instance, consider the scenario where a malicious party receives the shares of a certified input held by an honest party, which is done via an authenticated secret-sharing protocol, however while running the MPC itself it chooses to not use the shares received during the previously run authenticated secret-sharing protocol and uses an arbitrarily chosen share instead. The current definitions in [2,11] fails to safeguard against such an attack and would require additional assumptions to ensure the consistency of shares.

To be precise, the current protocol description of $\Pi_{\mathsf{CertInput}}$ in [2] (Section 5.1) emulates the authenticated secret-sharing, such that at the end of the protocol, if an input corresponds to a valid signature, the shares of that input is available to every party. This protocol first secret-shares the input, then using the shares held by everyone as input invokes another protocol Π_{Verify} to ascertain if the shares obtained in the previous phase corresponds to an input for which there is a valid signature. However, note that only Step 3 of Π_{Verify} considers the shares of the input, which need not be the shares used for running the MPC, unless additional consistency checks using the MACs on the shares are in place. Such details do not explicitly appear in the protocols presented in [2].

The protocols in [11] also follow a similar template based on authenticated secret-sharing. Their techniques consider two specific MPC protocols [20,21] for input certification. Concretely, Theorem 8 for input certification in [11] ensures that a malicious prover cannot feed an input which does not correspond to the valid signature. While it is not explicitly specified in [11] that the commitments to the inputs used for the batch verification of signatures are consistent with the inputs used for the remaining proof of knowledge statements, we assume that this is indeed the case.

In this paper, we recognize the benefits of having a formal definition to capture the consistency of shares of input used in authentication and the MPC. To this end, we explicitly provide an ideal functionality ensuring the same, and then present a construction satisfying this ideal functionality. We also avoid the possibility of using different inputs for certification and MPC by enforcing that the honest party shares must completely determine the reconstructed input which is being authenticated. While this observation has not been specified in either of the works, this specific restriction would also ensure that the consistency of shares holds for constructions in [2,11] as well.

We use efficient compressed DPoKs for signature verification instead of verifying signatures inside the MPC protocol, hence differing from both [2] and [11] in terms of techniques used and properties achieved. In particular, our compiler is modular, fully generic (works in a plug-and-play manner with any threshold linear secret sharing based MPC protocol), and avoids the (potentially expensive) protocol-specific techniques and pre-processing requirements that are inherent to [2,11]. Our compiler also enables stronger security guarantees as compared to abort security, namely identifiable abort (and even full security/guaranteed output delivery in certain cases), which neither [2] nor [11] achieves.

Distributed Zero-Knowledge. Various notions of distributed zero-knowledge have appeared in literature. The notion of distributed interactive proofs appeared in [42], in the context of relations describing the verification of signatures, where the signature is public and the secret key is shared. The notion in [50] considers a distributed prover in order to improve prover efficiency, but the witness is still held by one entity. In Feta [6], the distributed notion is a generalization of designated verifier to the threshold setting where a set of verifiers jointly verify the correctness of the proof. Prio [19] proposes secret shared non-interactive proofs where again, there is a single prover and many verifiers.

Our formulation of DPoKs also differs from recent works on distributed zkSNARKs [23,40,47], where the focus is on jointly computing a non-interactive publicly verifiable proof (with specific focus on Groth16 [33], Plonk [27] and Marlin [18]). Their constructions require additional interaction among the workers over private channels. On the other hand, we consider DPoKs where all interaction with the verifier takes place over a public broadcast channel. We also study the notion of *robust completeness* that guarantees completion even in the presence of malicious behavior while ensuring succinct proof size, which was not achieved in prior works. Note that distributed zkSNARKs fundamentally differ in their objective. DPOKs prove that the given shares (e.g., the one used for MPC) reconstruct a valid witness, whereas distributed zkSNARKs do not certify a given sharing.

A recent work on distributed zkSNARKs, called zkSaaS [31], considers a monolithic prover that aims outsources proof generation to (untrusted) servers in a privacy-preserving manner for increased efficiency. However, we target applications that require proving (algebraically structured) relations involving an already secret-shared witness. Plugging it naively does not work as a replacement for our proposed compiler since it would not ensure that the same input shares are used consistently in the authentication protocol and the core MPC. Additionally, similar to the distributed proofs with multiple verifier, [31] also requires expressing the algebraically structured relations as circuits, which is inefficient for the algebraic relations considered in our work.

Proofs on Secret-Shared Data. Notions of zero-knowledge proofs on distributed data is explored in recent works [6,13,34]. The former work proposes the abstraction of a fully linear PCP (FLPCP) where each verifier only has access to a share of the statement, and the latter work is based on MPC-in-the-head paradigm. The techniques of distributed verification [6,13,34] assumes the relations to be represented as an arithmetic circuit, whereas our DPoKs consider algebraic relations whose circuit representation is prohibitively expensive. Additionally, distributed verifier paradigm considers a designated prover who knows entire witness to create a proof oracle, which is verified in distributed fashion, while DPoKs do not require a prover which knows the entire witness. For example for proof of $g^x h^y = C$ wheres x and y belongs to different parties, a DPoK will succeed as long as provers have valid shares of x and y.

Our observation is that algebraic relations like discrete log is naturally distributed witness relation. A public statement and shared witness is better suited for algebraic relations, and our distributed zero-knowledge definition captures such natural relations. Since the focus of our work is on concrete efficiency (prover overhead, communication overhead), we take advantage of the algebraic nature of the relation to design concretely efficient DPoKs by modeling the witness as being distributed and statement being public. In this approach, we expect rich classes of protocols (compressed sigma protocols, Bulletproofs etc. that avoid circuit representation for several useful relations) to be amenable to be distributed under our definition. In addition, [13] provides sublinear communication only for special circuits (like degree 2) and the circuits of interest for us are unlikely to have this structure.

We also note that [13] does not consider the robustness property. We put forth the robustness notion that guarantees that the protocol runs to completion even in the presence of malicious parties (when the prover is honest). This property is indeed important for our applications, as this means that the compiled authenticated MPC protocol can identify malicious parties in the authentication stage. The distributed completeness guarantees of [6] considers robustness, however its protocol execution incurs communication cost linear in the size of the circuit in the offline phase. However, [6] does not allow aggregation of multiple instances of authentication of input into one execution of the underlying distributed protocol, which we support efficiently.

Finally, the motivating application for [13] is compiling passive security to active security, and therefore the statements that show up – like the next message function of the protocol – have a low degree circuit representation. We consider the authenticated input application where our relations of interest are algebraic in nature (e.g. verification of an algebraic signature scheme) and admit efficient sigma protocols.

1.4 Resistance to Known Vulnerabilities

Here, we present a discussion on why our proposed DPoK protocols and our compiler for authenticated MPC resist some known attacks and insecurities of ZKP protocols in practice.

Resistance to ROS Attacks. In [9], the authors presented an algorithm for solving ROS (Random inhomogeneities in a Overdetermined Solvable system of linear equations) $\mod p$ in polynomial time for $\ell > \log p$ dimensions, which leads to the ROS attack on certain advanced families of digital signatures which involve computations over secret shares. However, the ROS attack does not apply to our proposed DPoK protocols. In particular, note that the ROS attack only works when: (i) there are more than $\log p$ parallel sessions for the same shares, (ii) the adversary chooses its first message after seeing all of the other first messages from the honest parties, (iii) the adversary chooses the challenge.

The ROS attack is not applicable for our protocols as: (i) there are no parallel sessions in our protocols, (ii) each protocol is instantiated using the output of (the randomized) Share algorithm of the underlying secret sharing scheme $(\mathsf{Share}, \mathsf{Reconstruct})$, thereby ensuring that we do not reuse the shares across sessions, and in the round-efficient versions of our proposed protocols: (iii) the parties send non-interactive proofs instead of sending the first-messages separately (see $\Pi^{\mathsf{FS}}_{\mathsf{dlog}}$ in Appendix E of the full version of the paper [24]), and finally (iv) the challenge is not chosen by the adversary (verifier); it is determined by performing a hash of the available public transcript.

Resistance to OSNARK-Related Vulnerabilities. In [26], the authors provide a study of when SNARKs are insecure in the presence of certain oracles (in particular, the knowledge soundness guarantees do not hold in such settings since the extraction fails). As defined in [26], an OSNARK is a SNARK that guarantees extraction even in presence of an oracle for the prover. We note here that the negative result for the existence of OSNARKs, as outlined in [26], does not provide a general impossibility result, since it only applies either to SNARKs where the prover has access to oracles with secret states (such that the extractor does not have access to these states), and for standard-model SNARKs. We note that the attack does not apply: (i) to SNARKs in the ROM, and (ii) when the extractor is black-box in the adversary. Fiat-Shamir transformed Sigma protocols are also known to satisfy black-box *simulation-extractability*, i.e., knowledge soundness holds even in the presence of proof oracles [28,29]. Analogously, our Fiat-Shamir transformed round-efficient proofs of knowledge are simulation-extractable in the random oracle model, as we establish through formal proofs of security. In particular, there are no other oracles with secret states in our setting. We emphasize that signatures are already independently obtained by the parties on their inputs, and signing or signature-oracles are not included as part of our authenticated MPC protocols.

2 Preliminaries

In this section, we introduce notations and present preliminary background material. We refer to Appendixes B.1, B.2 and B.3 for additional preliminaries.

Notation. We write $x \leftarrow_R \chi$ to represent that an element x is sampled uniformly at random from a set/distribution $\mathcal{X}$. The output x of a deterministic algorithm $\mathcal{A}$ is denoted by $x = \mathcal{A}$ and the output x' of a randomized algorithm $\mathcal{A}'$ is denoted by $x' \leftarrow_R \mathcal{A}'$. For $n \in \mathbb{N}$, let $[n]$ denote the set $\{1, \dots, n\}$. For $a, b \in \mathbb{N}$ such that $a, b \geq 1$, we denote by $[a, b]$ the set of integers lying between a and b (both inclusive). We refer to $\lambda \in \mathbb{N}$ as the security parameter, and denote by $\mathsf{poly}(\lambda)$ and $\mathsf{negl}(\lambda)$ any generic (unspecified) polynomial function and negligible function in λ, respectively. A function $f : \mathbb{N} \to \mathbb{N}$ is said to be negligible in λ if for every positive polynomial p, $f(\lambda) < 1/p(\lambda)$ when λ is sufficiently large.

Let $\mathbb{G}$ be a group and $\mathbb{F}_p$ denote the field of prime order p. We use boldface to denote vectors. Let $\mathbf{g} = (g_1, \dots, g_n) \in \mathbb{G}^n$ and $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{F}_p^n$, then $\mathbf{g}^{\mathbf{x}}$ is defined by $\mathbf{g}^{\mathbf{x}} = g_1^{x_1} \cdots g_n^{x_n}$. For $\mathbf{g} = (g_1, \dots, g_n) \in \mathbb{G}^n$ and $\mathbf{h} = (h_1, \dots, h_n) \in \mathbb{G}^n$, $\mathbf{g} \circ \mathbf{h}$ denotes component-wise multiplication, and is defined by $\mathbf{g} \circ \mathbf{h} = (g_1 h_1, \dots, g_n h_n)$. For $\mathbf{g} = (g_1, \dots, g_n) \in \mathbb{G}^n$ and $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{F}_p^n$, $\mathbf{g}_L$ (similarly, $\mathbf{x}_L$) denotes the left half of the vector $\mathbf{g}(\mathbf{x})$ and $\mathbf{g}_R(\mathbf{x}_R)$ denotes the right half, such that $\mathbf{g} = \mathbf{g}_L \| \mathbf{g}_R$ and $\mathbf{x} = \mathbf{x}_L \| \mathbf{x}_R$.

2.1 Threshold Secret Sharing

For ease of exposition we define a special case of *threshold linear secret sharing* scheme below. For concreteness, the reader may assume a (t, n) Shamir Secret Sharing. The more general definition appears in Appendix C of the full version of the paper [24].

Definition 1 (Threshold Secret Sharing). *A (t, n) threshold secret sharing over finite field $\mathbb{F}$ consists of algorithms* $(\mathsf{Share}, \mathsf{Reconstruct})$ *as described below:*

- Share *is a randomized algorithm that on input $s \in \mathbb{F}$ samples a vector $(s_1, \dots, s_n) \in \mathbb{F}^n$, which we denote as $(s_1, \dots, s_n) \leftarrow_R \mathsf{Share}(s)$.*

- $\mathsf{Reconstruct}$ *is a deterministic algorithm that takes a set* $\mathcal{I} \subseteq [n]$, $|\mathcal{I}| \geq t$, *a vector* $(s_1, \ldots, s_{|\mathcal{I}|})$ *and outputs* $s = \mathsf{Reconstruct}((s_1, \ldots, s_{|\mathcal{I}|}), \mathcal{I}) \in \mathbb{F}$. *We will often omit the argument* $\mathcal{I}$ *when it is clear from the context.*

A threshold secret sharing scheme satisfies the following properties:

- **Correctness**: *For every* $s \in \mathbb{F}$, *any* $(s_1, \ldots, s_n) \leftarrow_R \mathsf{Share}(s)$ *and any subset* $\mathcal{I} = \{i_1, \ldots, i_q\} \subseteq [n]$ *with* $q > t$, *we have* $\mathsf{Reconstruct}((s_{i_1}, \ldots, s_{i_q}), \mathcal{I}) = s$.
- **Privacy**: *For every* $s \in \mathbb{F}$, *any* $(s_1, \ldots, s_n) \leftarrow_R \mathsf{Share}(s)$ *and any subset* $\mathcal{I} = \{i_1, \ldots, i_q\} \subseteq [n]$ *with* $q \leq t$, *the tuple* $(s_{i_1}, \ldots, s_{i_q})$ *is information-theoretically independent of* s.

A concrete (t, n) sharing scheme over a finite field $\mathbb{F}$, known as the Shamir Secret Sharing is realized by choosing a set of distinct points $\boldsymbol{\eta} = \{\eta_1, \ldots, \eta_n\}$ in $\mathbb{F}\backslash\{0\}$. Then given $s \in \mathbb{F}$, the Share algorithm uniformly samples a polynomial p of degree at most t such that $p(0) = s$ and outputs $(p(\eta_1), \ldots, p(\eta_n))$ as the shares. The $\mathsf{Reconstruct}$ algorithm essentially reconstructs the value $s = p(0)$ using Lagrangian interpolation. We canonically extend the Share and $\mathsf{Reconstruct}$ algorithms to vectors by applying them component-wise.

Definition 2 (Linear Code). *An* $[n, k, d]$*-linear code* $\mathcal{L}$ *over field* $\mathbb{F}$ *is a* k*-dimensional subspace of* $\mathbb{F}^n$ *such that* $d = \min\{\Delta(\mathbf{x}, \mathbf{y}) : \mathbf{x}, \mathbf{y} \in \mathcal{L}, \mathbf{x} \neq \mathbf{y}\}$. *Here* Δ *denotes the hamming distance between two vectors.*

We say that an $m \times n$ matrix $\mathbf{P} \in \mathcal{L}^m$ if each row of $\mathbf{P}$ is a vector in $\mathcal{L}$. We also overload the distance function Δ over matrices; for matrices $\mathbf{P}, \mathbf{Q} \in \mathbb{F}^{m \times n}$, we define $\Delta(\mathbf{P}, \mathbf{Q})$ to be the number of columns in which $\mathbf{P}$ and $\mathbf{Q}$ differ. For a matrix $\mathbf{P} \in \mathbb{F}^{m \times n}$ and an $[n, k, d]$ linear code $\mathcal{L}$ over $\mathbb{F}$, we define $\Delta(\mathbf{P}, \mathcal{L}^m)$ to be minimum value of $\Delta(\mathbf{P}, \mathbf{Q})$ where $\mathbf{Q} \in \mathcal{L}^m$.

Definition 3 (Reed Solomon code). *For any finite field* $\mathbb{F}$, *any* n*-length vector* $\boldsymbol{\eta} = (\eta_1, \ldots, \eta_n) \in \mathbb{F}^n$ *of distinct elements of* $\mathbb{F}$ *and integer* $k < n$, *the* Reed Solomon Code $\mathcal{RS}_{n,k,\boldsymbol{\eta}}$ *is an* $[n, k, n-k+1]$ *linear code consisting of vectors* $(p(\eta_1), \ldots, p(\eta_n))$ *where* p *is a polynomial of degree at most* $k-1$ *over* $\mathbb{F}$.

We note that shares output by (t, n) Shamir secret sharing are vectors in $[n, t+1, n-t]$ Reed Solomon code. We can leverage tests for membership of a vector in a linear code (based on parity-check matrix) to check if a set of shares $\{s_i\}_{i \in \mathcal{H}}$ for $\mathcal{H} \subseteq [n]$ and $|\mathcal{H}| > t$ uniquely determine a shared value s for Shamir Secret Sharing scheme. Below, we formalise the notion of consistent shares and state a lemma to check such shares. In the interest of space, we directly state the results for general $m \in \mathbb{N}$, i.e. when vectors $\mathbf{s} \in \mathbb{F}^m$ are shared.

Definition 4 (Consistent Shares). *Let* $\mathcal{L}$ *be the linear code determined by a* (t, n) *Shamir secret sharing scheme over finite field* $\mathbb{F}$. *For* $m \in \mathbb{N}$, *we call a set of shares* $\{\mathbf{s}_i\}_{i \in \mathcal{H}}$ *for* $\mathcal{H} \subseteq [n]$ *with* $|\mathcal{H}| \geq t+1$ *to be* $\mathcal{L}^m$-consistent *if there exists* $(\mathbf{v}_1, \ldots, \mathbf{v}_n) \in \mathcal{L}^m$ *such that* $\mathbf{s}_i = \mathbf{v}_i$ *for* $i \in \mathcal{H}$. *In this case* $\mathbf{s} =$

$\mathsf{Reconstruct}(\mathbf{v}_1, \ldots, \mathbf{v}_n) \in \mathbb{F}^m$ *is the unique shared value determined by the shares* $\{\mathbf{s}_i\}_{i \in \mathcal{H}}$.

We define the predicate $\mathsf{Consistent} : \mathbb{F}^{\mathcal{H}+1} \to \{0, 1\}$ *as*

$$\mathsf{Consistent}(\{\mathbf{s}_i\}_{i\in\mathcal{H}}, \mathbf{s}) = \begin{cases} 1, & |\mathcal{H}| \le t \\ 1, & |\mathcal{H}| > t \wedge \{\mathbf{s}_i\}_{i\in\mathcal{H}} \text{ is } \mathcal{L}^m\text{-consistent} \\ & \quad \bigwedge \mathsf{Reconstruct}(\{\mathbf{s}_i\}_{i\in\mathcal{H}}) = \mathbf{s} \\ 0, & \text{otherwise.} \end{cases}$$

We use this $\mathsf{Consistent}(.)$ predicate to determine if a vector $\mathbf{s}$ can be a possible candidate which could have been used to generate the set of shares held by the honest parties $\{\mathbf{s}_i\}_{i\in\mathcal{H}}$.

Lemma 1. *Let $\mathcal{L}$ be the linear code determined by a (t, n) Shamir secret sharing scheme over finite field $\mathbb{F}$. Then for $m \in \mathbb{N}$ and all $\mathcal{H} \subseteq [n]$ with $q = |\mathcal{H}| \ge t+1$, there exists $q \times (n-t)$ matrix $\mathbf{H}_{\mathcal{H}}\mathcal{H}$ over $\mathbb{F}$ such that shares $\{\mathbf{s}_i\}_{i\in\mathcal{H}}$ are $\mathcal{L}^m$-consistent and determine the value $\mathbf{s} \in \mathbb{F}^m$ if and only if $\mathbf{X}\mathbf{H}_{\mathcal{H}} = (\mathbf{s}, \mathbf{0}^{n-t-1})$ where $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_q)$ is some canonical ordering of $\{\mathbf{s}_i\}_{i\in\mathcal{H}}$.*

Proof. We sketch the proof. For a matrix $\mathbf{P} \in \mathcal{L}^m$, we have $\mathbf{PH} = \mathbf{0}^{n-t-1}$ where $\mathbf{H}$ is the parity check matrix for the $[n, t+1, n-t]$ code $\mathcal{L}$. Now for $\mathcal{H} \subseteq [n]$ with $|\mathcal{H}| \ge t+1$, and matrix $\mathbf{X}$ determined by $\mathcal{L}^m$-consistent shares $(\mathbf{s}_i)_{i\in\mathcal{H}}$, there exists a matrix $\mathbf{T}_{\mathcal{H}}$ such that $\mathbf{X}\mathbf{T}_{\mathcal{H}} \in \mathcal{L}^m$, and hence $\mathbf{X}\mathbf{T}_{\mathcal{H}}\mathbf{H} = \mathbf{0}^{n-t-1}$. Thus for $\mathbf{H}_{\mathcal{H}} = [\mathbf{k}, \mathbf{T}_{\mathcal{H}}\mathbf{H}]$ where $\mathbf{k}$ is the column of reconstruction coefficients for the set $\mathcal{H}$, we have $\mathbf{X}\mathbf{H}_{\mathcal{H}} = (\mathbf{s}, \mathbf{0}^{n-t-1})$.

2.2 Proofs of Knowledge

Let $\mathcal{R}$ be a NP-relation and $\mathcal{L}$ be the corresponding NP-language, where $\mathcal{L} = \{x : \exists\ w \text{ such that } (x, w) \in \mathcal{R}\}$. Here, x is called an *instance or statement* and w is called a *witness*. An *interactive proof system* consists of a pair of PPT algorithms $(\mathcal{P}, \mathcal{V})$. $\mathcal{P}$, known as the prover algorithm, takes as input an instance $x \in \mathcal{L}$ and its corresponding witness w, and $\mathcal{V}$, known as the verifier algorithm, takes as input an instance x. Given a public instance x, the prover $\mathcal{P}$, convinces the verifier $\mathcal{V}$, that $x \in \mathcal{L}$. At the end of the protocol, based on whether the verifier is convinced by the prover's claim, $\mathcal{V}$ outputs a decision bit. A stronger *proof of knowledge* (PoK)[5] property says that if the verifier is convinced, then the prover knows a witness w such that $(x, w) \in \mathcal{R}$. In this paper, we consider POKs that satisfy two security properties, namely, *honest-verifier zero-knowledge* (HVZK) and *special-soundness.*

A protocol is said to be *honest-verifier zero-knowledge* (HVZK) if the transcript of messages resulting from a run of the protocol can be simulated by an efficient algorithm without knowledge of the witness. A protocol is said to

[5] Throughout this paper, we use *proof* and *argument* interchangeably, but we are only concerned with arguments (proofs with computational soundness) in this paper.

have *k-special-soundness*, if given k accepting transcripts, an extractor algorithm can output a w' such that $(x, w') \in \mathcal{R}$. Furthermore, a protocol is said to have $(k_1, \ldots, k_\mu)$*-special-soundness* [14], if given a tree of $\prod_{i=1}^{\mu} k_i$ accepting transcripts, the extractor can extract a valid witness. Here, each vertex in the tree of $\prod_{i=1}^{\mu} k_i$ accepting transcripts corresponds to the prover's messages and each edge in the tree corresponds the verifier's challenge, and each root-to-leaf path is a transcript. An interactive protocol is said to be *public-coin* if the verifier's messages are uniformly random strings. Public-coin protocols can be transformed into non-interactive arguments using the Fiat-Shamir [25] heuristic by deriving the verifier's messages as the output of a Random Oracle. In this work, we consider public-coin protocols.

We refer to Appendix B.1 of the full version of the paper [24] for a detailed treatment of non-interactive zero-knowledge (NIZK) proof systems.

2.3 BBS+ Signatures and PoK for BBS

In this section, we recall the BBS+ signature scheme [12,15,39], and its proof of knowledge. We use the variant of BBS+ signatures and the proof of knowledge from [15], which is the currently adopted variant in the IETF standard for verifiable crendentials [39]. Later, we also describe a slight variant of the BBS+ proof of knowledge from [15], which leads to corresponding distributed proofs with better amortized complexity (i.e., when several DPoKs are required at a time).

Definition 5 (BBS+ Signature Scheme [12,39]). *The BBS+ signature scheme to sign a message of the form* $\mathbf{m} = (m_1, \ldots, m_\ell) \in \mathbb{F}_p^\ell$ *consists of a tuple of PPT algorithms* ($\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify}$) *described as follows :*

- $\mathsf{Setup}(1^\lambda)$ *: For security parameter* λ*, this algorithm outputs groups* $\mathbb{G}_1, \mathbb{G}_2$*, and* $\mathbb{G}_T$ *of prime order* p*, with an efficient bilinear map* $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ *as part of the public parameters* pp*, along with* g_1 *and* g_2*, which are the generators of groups* $\mathbb{G}_1$ *and* $\mathbb{G}_2$ *respectively.*
- $\mathsf{KeyGen}(\mathsf{pp})$ *: This algorithm samples* $(h_0, \ldots, h_\ell) \leftarrow_R \mathbb{G}_1^{\ell+1}$ *and* $x \leftarrow_R \mathbb{F}_p^*$*, computes* $w = g_2^x$ *and outputs* $(\mathsf{sk}, \mathsf{pk})$*, where* $\mathsf{sk} = x$ *and* $\mathsf{pk} = (g_1, w, h_0, \ldots, h_\ell)$*.*
- $\mathsf{Sign}(\mathsf{sk}, m_1, \ldots, m_\ell)$ *: This algorithm samples* $\beta, s \leftarrow_R \mathbb{F}_p$*, computes* $A = \left(g_1 h_0^s \prod_{i=1}^{\ell} h_i^{m_i}\right)^{\frac{1}{\beta+x}}$ *and outputs* $\sigma = (A, \beta, s)$*.*
- $\mathsf{Verify}(\mathsf{pk}, (m_1, \ldots, m_\ell), \sigma)$ *: This algorithm parses* σ *as* $(\sigma_1, \sigma_2, \sigma_3)$*, and checks*
$$e\left(\sigma_1, w g_2^{\sigma_2}\right) = e\left(g_1 h_0^{\sigma_3} \prod_{i=1}^{\ell} h_i^{m_i},\ g_2\right).$$
If yes, it outputs 1, and outputs 0 otherwise.

PoK for BBS+ Signature Scheme. We present a modified proof of knowledge (PoK) for BBS+ signatures, building on the PoK originally proposed in [15] (summarized in Appendix B.3 of the full version of the paper [24]), wherein we split the relation $d^{-r_3} h_0^{s'} \prod_{i=1}^{\ell} h_i^{m_i} = g_1^{-1}$ by requiring the prover to equivalently show:

$$d^{-r_3} h_0^{s'-\eta} = C \wedge h_0^{\eta} \prod_{i=1}^{\ell} h_i^{m_i} = D \wedge C \cdot D = g_1^{-1}$$

The above decomposition has advantage that the (long) message $\mathbf{m}$ appears only with public generators which leads to better aggregation of DPoKs over several messages. The complete modified protocol appears below.

– **Common Input**: Public Key $\mathsf{pk} = (w, h_0, \ldots, h_\ell)$
– **$\mathcal{P}$'s inputs**: Message $\mathbf{m} \in \mathbb{F}_p^{\ell}$ and signature $\sigma = (A, \beta, s)$ on $\mathbf{m}$, with $A = \left(g_1 h_0^s \prod_{i=1}^{\ell} h_i^{m_i}\right)^{\frac{1}{\beta+x}}$.
  1. $\mathcal{P}$ samples $r_1 \leftarrow_R \mathbb{F}_p^*$ and computes $A' = A^{r_1}$ and $r_3 = r_1^{-1}$
  2. $\mathcal{P}$ computes $\bar{A} = (A')^{-\beta} \cdot b^{r_1}$, where $b = g_1 h_0^s \prod_{i=1}^{\ell} h_i^{m_i}$.
  3. $\mathcal{P}$ samples $r_2 \leftarrow_R \mathbb{F}_p$ and computes $d = b^{r_1} \cdot h_0^{-r_2}$ and $s' = s - r_2 \cdot r_3$
  4. $\mathcal{P}$ samples $\eta \leftarrow_R \mathbb{F}_p$ and sets $C = d^{-v} h_0^{s'-\eta}$, and $D = h_0^{\eta} \prod_{i=1}^{\ell} h_i^{m_i}$.
  5. $\mathcal{P}$ sends $(A', \bar{A}, d, C, D)$ to $\mathcal{V}$.
  6. $\mathcal{P}$ and $\mathcal{V}$ run a ZKPoK for the discrete-logarithm relation:
$$(A')^{-\beta} h_0^{r_2} = \bar{A}/d \wedge d^{-r_3} h_0^{s'-\eta} = C \wedge h_0^{\eta} \prod_{i=1}^{\ell} h_i^{m_i} = D$$
     where $(\mathbf{m}, r_2, r_3, \beta, s', \eta)$ is the witness.
  7. $\mathcal{V}$ checks that $A' \neq 1_{\mathbb{G}_1}, C \cdot D = g_1^{-1}, e(A', w) = e(\bar{A}, g_2)$, verifies the ZKPoK proof and outputs 1 if all the checks pass, and 0 otherwise.

3 Distributed Proof of Knowledge

In this section, we formalize the notion of *distributed* proof of knowledge (DPoK) in which multiple provers, each having a share of the witness engage in an interactive protocol with a verifier to convince it that their shares determine a valid witness. The provers do not directly interact with each other, and all the interaction with the verifier takes place over a public broadcast channel.

3.1 Defining a DPoK

Definition 6 (Distributed Proof of Knowledge.) We define n-party *distributed proof of knowledge* for relation generator RGen and a secret-sharing scheme $\mathsf{SSS} = (\mathsf{Share}, \mathsf{Reconstruct})$ by the tuple $\mathsf{DPoK}_{\mathsf{SSS},\mathsf{RGen}} = (\mathsf{Setup}, \Pi)$ where Setup is a PPT algorithm and Π is an interactive protocol between PPT algorithms $\mathcal{P}$ (prover), $\mathcal{V}$ (verifier) and $\mathcal{W}_1, \ldots, \mathcal{W}_n$ (workers) defined as follows:

- **Setup Phase**: For relation $\mathcal{R} \leftarrow_R \mathsf{RGen}(1^\lambda)$, $\mathsf{Setup}(\mathcal{R})$ outputs public parameters pp as $\mathsf{pp} \leftarrow_R \mathsf{Setup}(\mathcal{R})$. The setup phase is required to be executed only once for a given relation $\mathcal{R}$. We assume $\mathcal{R}$ consists of pairs $(\mathbf{x}, \mathbf{w})$ where $\mathbf{w}$ is parsed as $(\mathbf{s}, \mathbf{t}))$ with $\mathbf{s} \in \mathbb{F}^m$. Looking ahead, we partition the witness as $(\mathbf{s}, \mathbf{t})$ to explicitly specify which parts of the witness later needs to be shared[6].
- **Input Phase**: The prover $\mathcal{P}$ receives $(\mathbf{x}, (\mathbf{s}, \mathbf{t})) \in \mathcal{R}$ as input, while the worker $\mathcal{W}_i$, $i \in [n]$ receives $(\mathbf{x}, \mathbf{s}_i)$ as input, where $(\mathbf{s}_1, \ldots, \mathbf{s}_n) \leftarrow_R \mathsf{Share}(\mathbf{s})$. All parties receive $\mathbf{x}$ as input.
- **Preprocessing Phase**: This is (an optional) phase where the prover $\mathcal{P}$ sends some auxiliary information aux_i to worker $\mathcal{W}_i$ using secure private channels.
- **Interactive Phase**: In this phase, the parties interact using a public broadcast channel according to the protocol Π. The protocol Π is a k-round protocol for some $k \in \mathbb{N}$, with $(\mathsf{pp}, \mathbf{x}, \mathbf{s}, \mathbf{t})$ as $\mathcal{P}$'s input, $(\mathsf{pp}, \mathbf{x}, \mathbf{s}_i, \mathsf{aux}_i)$ as the input of $\mathcal{W}_i$ and $(\mathsf{pp}, \mathbf{x})$ as the input of $\mathcal{V}$. The verifier's message in each round $j \in [k]$ consists of a uniformly sampled challenge $\mathbf{c}_j \in \mathbb{F}^{\ell_j}$ for $\ell_j \in \mathbb{N}$. In each round $j \in [k]$, the worker $\mathcal{W}_i$ (resp. the prover $\mathcal{P}$) broadcasts a message $\mathbf{m}_{ij}$ (resp., $\mathbf{m}_i$) which depends on it's random coins and the messages received in prior rounds (including pre-processing phase).
- **Output Phase**: At the conclusion of k rounds, verifier outputs a bit $b \in \{0, 1\}$ indicating accept (1) or reject (0).

A distributed proof of knowledge $\mathsf{DPoK}_{\mathsf{SSS},\mathsf{RGen}}$ as described above is said to be *t-private*, *ℓ-robust* if the following hold:

- **Completeness**: We say that completeness holds if for all $\mathcal{R} \leftarrow_R \mathsf{RGen}(1^\lambda)$ and $(\mathbf{x}, \mathbf{s}) \in \mathcal{R}$, the honest execution of all the phases results in 1 being output in the output phase with probability 1.
- **Knowledge-Soundness**: We say that knowledge soundness holds if for any PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_2$ corrupts the prover $\mathcal{P}$ and subset of workers $\{\mathcal{W}_i\}_{i\in\mathsf{C}}$ for some $\mathsf{C} \subseteq [n]$, there exists an extractor Ext with oracle access to $\mathcal{A}_2$ (recall that the prover and the set of corrupt $\mathcal{W}_i$ are controlled by $\mathcal{A}_2$) such the following probability is negligible.

$$\Pr\left[\begin{array}{c|r} \mathcal{V}_{\mathcal{A},\Pi}(\mathsf{pp}, \mathbf{x}) = 1 \wedge & \mathcal{R} \leftarrow_R \mathsf{RGen}(\lambda) \\ ((\mathbf{x}, (\mathbf{s}, \mathbf{t})) \notin \mathcal{R} \vee & \mathsf{pp} \leftarrow_R \mathsf{Setup}(\mathcal{R}) \\ \mathsf{Consistent}(\{\mathbf{s}_i\}_{i\notin\mathsf{C}}, \mathbf{s}) = 0) & (\mathbf{x}, \{\mathbf{s}_i\}_{i\notin\mathsf{C}}) \leftarrow_R \mathcal{A}_1(\mathsf{pp}) \\ & (\mathbf{s}, \mathbf{t}) \leftarrow_R \mathsf{Ext}^{\mathcal{A}_2}(\mathsf{pp}, \mathbf{x}, \{\mathbf{s}_i\}_{i\notin\mathsf{C}}) \end{array}\right]$$

In the above, $\mathcal{V}_{\mathcal{A},\Pi}(\mathsf{pp}, \mathbf{x})$ denotes the verifier's output in the protocol Π with its input as $(\mathsf{pp}, \mathbf{x})$ and $\mathcal{A}$ being the adversary. The extractor takes as input the shares of the honest parties specified by the adversary $\mathcal{A}_1$, and with all but negligible probability extracts a valid witness.

[6] We specify $\mathbf{s} \in \mathbb{F}^m$ since our secret sharing works over a finite field. The witness component $\mathbf{t}$ need not, in general, be a field element. In fact, in our application, the witness is a message signature pair where the message is in $\mathbb{F}^m$ and the signature is a group element. This group element is not secret shared, yet, the DPOK guarantees extraction of a valid signature message pair.

- **Honest Verifier Zero-Knowledge**: We say that a DPoK is honest verifier zero-knowledge if for all $\mathcal{R} \leftarrow_R \mathsf{RGen}(1^\lambda)$, $(\mathbf{x}, \mathbf{s}) \in \mathcal{R}$ and any PPT adversary $\mathcal{A}$ corrupting a set of workers $\{\mathcal{W}_i\}_{i \in \mathsf{C}}$, where $|\mathsf{C}| \leq t$, there exists a PPT simulator Sim such that $\mathsf{View}_{\mathcal{A},\Pi}(\mathsf{pp}, \mathbf{x})$ is indistinguishable from $\mathsf{Sim}(\mathsf{pp}, \mathbf{x})$ for $\mathsf{pp} \leftarrow_R \mathsf{Setup}(\mathcal{R})$. Here, the view is given by $\mathsf{View}_{\mathcal{A},\Pi} = \{\mathbf{r}, (\mathbf{M}_i)_{i \in \mathsf{C}}\}$ where $\mathbf{r}$ denotes the internal randomness of $\mathcal{A}$ and $\mathbf{M}_i$ is the set of all messages received by $\mathcal{W}_i$ in Π. We remark that we define honest-verifier zero-knowledge as is standard for public-coin interactive protocols. After Fiat-Shamir compilation into a non-interactive proof, we get full zero-knowledge against a malicious verifier.
- **Robust-Completeness**: We say that robust-completeness holds if for all $\mathcal{R} \leftarrow_R \mathsf{RGen}(1^\lambda)$, $(\mathbf{x}, \mathbf{s}) \in \mathcal{R}$ and any PPT adversary $\mathcal{A}$ corrupting a set of workers $\{\mathcal{W}_i\}_{i \in \mathsf{C}}$, where $|\mathsf{C}| \leq \ell$, $\mathcal{V}_{\mathcal{A},\Pi}(\mathsf{pp}, \mathbf{x}) = 1$ with overwhelming probability where $\mathsf{pp} \leftarrow_R \mathsf{Setup}(\mathcal{R})$.

Remark 1. Robust completeness is a stronger notion of completeness in the sense that it holds even if some corrupt workers deviate maliciously from the protocol, as opposed to the standard notion of completeness which only holds if all the workers follow the protocol. Looking ahead, we use robust complete DPoKs to design authenticated MPC protocols that preserve the underlying protocol's resilience against malicious behavior.

Remark 2. We assume that the sharing phase is executed before the onset of DPoK, hence the knowledge soundness extractor of DPoK expects honest party shares in order to extract the witness. Since knowledge soundness is supposed to hold against a corrupt prover and some corrupt workers, it is meaningful to say that the adversary breaks knowledge soundness if no extractor can construct corrupt party shares that **together with the honest party shares** determine a valid witness. Note that extractor is required to produce shares of corrupt parties which "explain" the successful outcome of the protocol in conjunction with the shares used by honest parties. Hence, DPoK enables us to certify a given sharing.

Remark 3. We assume an honest verifier $\mathcal{V}$ for ease of exposition. In Section E of the full version of the paper [24], we relax this assumption by transforming any $\mathsf{DPoK}_{\mathsf{SSS},\mathsf{RGen}}$ protocol that uses only public coins and communication over broadcast channels between the workers and the verifier (with no communication among the workers), into a round-efficient version $\mathsf{RE\text{-}DPoK}_{\mathsf{SSS},\mathsf{RGen}}$ in the random oracle model, wherein the verifier's challenge is computed using the Fiat-Shamir heuristic [25].

3.2 Robust Complete DPoK for Discrete Log

In this section, we provide a $\mathsf{DPoK}_{\mathsf{SSS},\mathsf{DlogGen}}$ for the discrete log relation based on Shamir Secret Sharing (SSS) [49]. Let $\mathsf{DlogGen}$ be a relation generator that on input $(1^\lambda, 1^\ell)$ outputs $(\mathbb{G}, \mathbf{g}, p)$ where p is a λ-bit prime, $\mathbb{G}$ is a cyclic group of order p and $\mathbf{g} = (g_1, \ldots, g_\ell) \leftarrow_R \mathbb{G}^\ell$ is a uniformly sampled set of generators.

The associated relation $\mathcal{R}^{\mathrm{DL}}$ is defined by $(z, \mathbf{s}) \in \mathcal{R}^{\mathrm{DL}}$ if $\mathbf{g}^{\mathbf{s}} = z$. Let $\mathsf{SSS} = (\mathsf{Share}, \mathsf{Reconstruct})$ denote (t, n) Shamir secret sharing over $\mathbb{F}_p$. Our protocol Π_{dlog} realizing $\mathsf{DPoK}_{\mathsf{SSS},\mathsf{DlogGen}}$ is as below. However, for ease of exposition, we first explain a simpler non-robust version of the protocol, before explaining the robust version. We use an instantiation of compressed sigma protocols (CSP) due to Attema et al. [3] as a black-box (see Appendix B.2 of the full version of the paper [24] for details). We run CSP protocol instances over a broadcast channel, meaning that each worker $\mathcal{W}_i$ (playing the role of the prover of that instance) broadcasts its messages as part of the CSP protocol, and the verifier broadcasts all challenges as well. [7]

Warm-Up: Non-robust DPoK for DLOG. We begin by describing a simpler, non-robust version of Π_{dlog} outlined above, which we call $\Pi_{\mathsf{nr\text{-}dlog}}$. Let us consider the scenario where the parties $\mathcal{W}_i$, $i \in [n]$, holds the shares $\mathbf{s}_i$ for a secret $\mathbf{s}$ such that $(z, \mathbf{s}) \in \mathcal{R}^{\mathrm{DL}}$, i.e. $z = \mathbf{g}^{\mathbf{s}}$. Now note that since $(\mathbf{s}_1, \ldots, \mathbf{s}_n) \leftarrow_R \mathbf{s}$, there exists some publicly known k_i such that $\sum_i k_i \mathbf{s}_i = \mathbf{s}$. In particular, the protocol $\Pi_{\mathsf{nr\text{-}dlog}}$ executes the following steps:

- **Input Phase:** The prover holds $(z, \mathbf{s})$ and each worker $\mathcal{W}_i$ $(i \in [n])$ holds $(z, \mathbf{s}_i)$, where $\mathbf{s}_i$ are shares of $\mathbf{s}$ i.e. $(\mathbf{s}_1, \ldots, \mathbf{s}_n) \leftarrow_R \mathsf{Share}(\mathbf{s})$.

<u>**Interactive Phase**</u>

- Each worker $\mathcal{W}_i$ $(i \in [n])$ broadcasts a commitment $A_i = \mathbf{g}^{\mathbf{s}_i}$ to their shares $\mathbf{s}_i$, along with a proof of knowledge π_i of its exponent $\mathbf{s}_i$ with respect to the associated commitment A_i.
- Thereafter, the verifier checks the following:
  - · The proofs π_i (with respect to the broadcast A_i) are valid for all $i \in [n]$.
  - · The broadcast A_i and the publicly known z satisfies the relation $z = \prod_i A_i^{k_i}$ for the publicly known reconstruction coefficients $\{k_i : i \in [n]\}$.

Robust DPoK for DLOG. Note that the previously described protocol $\Pi_{\mathsf{nr\text{-}dlog}}$ achieves completeness only if all of the parties participating to produce the proof are honest. To achieve completeness even in the presence of corrupt parties, known as the stronger guarantee of robust completeness, we require error-correction. However the shares that requires error-correction are in the exponent of a publicly known group element and it is known from [43] that error correction is not possible in the exponent. To ensure that error correction of the shares present in the exponent is possible, we reveal a random linear combination of the codewords and leverage the coding theoretic lemma that states that a random linear combination of a set of codewords from an error-correcting code (e.g., Reed-Solomon code) retains the position of errors as long as the number of errors are *small*. In particular, the protocol Π_{dlog} executes the following steps:

- **Input Phase:** The prover holds $(z, \mathbf{s})$ and each worker $\mathcal{W}_i$ $(i \in [n])$ holds $(z, \mathbf{s}_i)$, where $\mathbf{s}_i$ are shares of $\mathbf{s}$ i.e. $(\mathbf{s}_1, \ldots, \mathbf{s}_n) \leftarrow_R \mathsf{Share}(\mathbf{s})$.

[7] Note that here the witness is $\mathbf{s} \in \mathbb{F}_p^{\ell}$, and we do not have any component $\mathbf{t}$ which is not being secret-shared.

- **Pre-processing:** We need an additional preprocessing step for providing robustness. In this phase, before the onset of the interactive phase of the protocol, the prover samples $r \leftarrow_R \mathbb{F}_p$, computes $(r_1, \ldots, r_n) \leftarrow_R \mathsf{Share}(r)$ and sends the share r_i to the worker $\mathcal{W}_i$.

Interactive Phase

- **Commit to Shares:** In the interactive phase, each worker $\mathcal{W}_i$ ($i \in [n]$) first commit to their respective shares by
  - broadcasting $A_i = \mathbf{g}^{\mathbf{s}_i}$ and running its associated proof of knowledge $\mathsf{CSP}\{(A_i, \mathbf{s}_i) : \mathbf{g}^{\mathbf{s}_i} = A_i\}$ over broadcast to obtain π_{i1}.
  - broadcasting $B_i = h_1^{r_i} h_2^{\omega_i}$ for $\omega_i \leftarrow_R \mathbb{F}_p$ and running its its associated proofs of knowledge $\mathsf{CSP}\{(B_i, (r_i, \omega_i)) : h_1^{r_i} h_2^{\omega_i} = B_i\}$ over broadcast to obtain π_{i2}.
- **Reveal Linear Form over Shares:** The verifier samples a challenge $\boldsymbol{\gamma} \leftarrow_R \mathbb{F}_p^\ell$ and broadcasts it. Thereafter, the workers broadcast the linear form $v_i = \langle \boldsymbol{\gamma}, \mathbf{s}_i \rangle + r_i$. Recall that, we know that random linear combination of a codeword is also a codeword (recalled in Lemma 2 of the full version of the paper [24]). Using Lemma 2 [24], since $\{(\mathbf{s}_i, r_i) : i \in [n]\}$ are codewords respectively, the linear combination of those codewords $(v_1, \ldots, v_n)$ using the randomly sampled γ is also a codeword.
  Additionally, to ensure that corrupt workers use $\mathbf{s}_i, r_i$ consistent with earlier commitments A_i, B_i we additionally require them to run the following proof of knowledge CSP over broadcast to obtain π_{i3}:

$$\pi_{i3} = \mathsf{CSP}\{((A_i B_i, \boldsymbol{\gamma} \| \mathbf{1} \| \mathbf{0}, v_i), (\mathbf{s}_i, r_i, \omega_i)) : \mathbf{g}^{\mathbf{s}_i} h_1^{r_i} h_2^{\omega_i} = A_i B_i \wedge \langle \boldsymbol{\gamma}, \mathbf{s}_i \rangle + r_i = v_i\}.$$

- **Verifier Determines Honest Commitments:** Let $\mathbf{v} = (v_1, \ldots, v_n)$, defined by $v_i = \langle \boldsymbol{\gamma}, \mathbf{s}_i \rangle + r_i$, be the vector of honestly computed values, and $\mathbf{v}' = (v'_1, \ldots, v'_n)$ be the respective broadcast values received by the workers in the previous step. If one of the proofs π_{i1}, π_{i2} or π_{i3} is invalid, the verifier set $b_i = 0$ else it sets $b_i = 1$. Since $\Delta(\mathbf{v}', \mathbf{v}) \leq d < (n-t)/2$, $\mathcal{V}$ can compute $\mathbf{v}$ from $\mathbf{v}'$ by decoding algorithm (e.g. Berlekamp-Welch) for Reed-Solomon codes. Set $\mathsf{C} = \{i \in [n] : v_i \neq v'_i \vee b_i = 0\}$ and let $\mathbf{H}_Q = (h_{jk})$ denote the matrix guaranteed by Lemma 1 for $Q = [n] \backslash \mathsf{C} = \{i_1, \ldots, i_q\}$ for $q \in \mathbb{N}$.
  Informally, C is the set consisting of the position of 'errors' noted by the verifier and the new reconstruction coefficient k'_i is computed for the set $[n] \setminus \mathsf{C} = \{i_1, \ldots, i_q\}$. Thereafter the verifier proceeds with the final check with the non-error positions in $\{i_1, \ldots, i_q\}$ by using the new reconstruction coefficients and the corresponding commitments sent in the previous round. Also, we rely on the fact that we use shares of a codeword $(\mathbf{s}, r)$ in the proof of knowledge π_{i3} to ensure that the received values $(v_1, \ldots, v_n)$, if correctly computed, would also be a codeword and error-correction can be used on the new codeword $(v_1, \ldots, v_n)$.
- **Output using Honest Messages:** $\mathcal{V}$ outputs $(1, \mathsf{C})$ if $\left(\prod_{j \in [q]} A_{i_j}^{h_{jk}}\right)_{k=1,\ldots,n-t} = (z, \mathbf{0}^{n-t-1})$, and $(0, \{\mathcal{P}\})$ otherwise.

This is achieved via the additional steps (4b) through (6) in Π_{dlog} outlined in the figure above. We subsequently present a formal proof that Π_{dlog} achieves d-robust completeness for $d < \mathsf{dist}/2$, where $\mathsf{dist} = (n-t)$ is the minimum distance of the Reed-Solomon code induced by (t, n)-SSS.

Remark 4. The final step of protocol Π_{dlog} checks $(n-t)$ equations over exponents and not just the reconstruction equation. This is to ensure that we extract the witness consistent with honest party shares of the witness. This is crucial in the security proof of our compiler for honest majority protocols where honest party shares determine a unique consistent witness, and this ensures that corrupt parties use the same inputs in both the DPoK protocol and the associated MPC protocol.

Protocol Π_{dlog}

1. **Public Parameters**: Let $(\mathbb{G}, \mathbf{g}, p) \leftarrow_R \mathsf{DlogGen}(1^\lambda, 1^\ell)$. Let $\mathcal{R}^{\mathrm{DL}}$ denote the relation consisting of pairs $(z, \mathbf{s})$ such that $\mathbf{g}^{\mathbf{s}} = z$. Let $(h_1, h_2) \leftarrow_R \mathsf{Setup}(\mathcal{R}^{\mathrm{DL}})$ be two independent generators of $\mathbb{G}$.
2. **Input Phase**: The prover gets $(z, \mathbf{s})$ while workers $\mathcal{W}_i$, $i \in [n]$ are given $(z, \mathbf{s}_i)$ where $(\mathbf{s}_1, \ldots, \mathbf{s}_n) \leftarrow_R \mathsf{Share}(\mathbf{s})$. [8]
3. **Pre-processing:** Prover samples $r \leftarrow_R \mathbb{F}_p$, computes $(r_1, \ldots, r_n) \leftarrow_R \mathsf{Share}(r)$ and sends r_i to $\mathcal{W}_i$ for $i \in [n]$.
4. **Commit to Shares:** In the interactive phase, each worker $\mathcal{W}_i$, for $i \in [n]$, does the following.
   (a) $\mathcal{W}_i$ broadcasts $A_i = \mathbf{g}^{\mathbf{s}_i}$ and runs its associated proofs of knowledge $\mathsf{CSP}\{(A_i, \mathbf{s}_i) : \mathbf{g}^{\mathbf{s}_i} = A_i\}$ over broadcast to obtain π_{i1}.
   (b) $\mathcal{W}_i$ broadcasts $B_i = h_1^{r_i} h_2^{\omega_i}$ for $\omega_i \leftarrow_R \mathbb{F}_p$ and runs its associated proofs of knowledge $\mathsf{CSP}\{(B_i, (r_i, \omega_i)) : h_1^{r_i} h_2^{\omega_i} = B_i\}$ over broadcast to obtain π_{i2}.
5. **Reveal Linear Form over Shares:**
   (a) $\mathcal{V}$ samples $\boldsymbol{\gamma} \leftarrow_R \mathbb{F}_p^\ell$ and broacasts it.
   (b) For all $i \in [n]$, $\mathcal{W}_i$ computes $v_i = \langle \boldsymbol{\gamma}, \mathbf{s}_i \rangle + r_i$ and broadcasts v_i.
   (c) For all $i \in [n]$, $\mathcal{W}_i$ also runs the associated proof of knowledge to obtain π_{i3}, i.e.
$$\pi_{i3} = \mathsf{CSP}\{((A_i B_i, \boldsymbol{\gamma} \| \mathbf{1} \| \mathbf{0}, v_i), (\mathbf{s}_i, r_i, \omega_i)) : \\ \mathbf{g}^{\mathbf{s}_i} h_1^{r_i} h_2^{\omega_i} = A_i B_i \wedge \langle \boldsymbol{\gamma}, \mathbf{s}_i \rangle + r_i = v_i\}.$$
6. **Verifier Determines Honest Commitments:**
   (a) Let $\mathbf{v}' = (v_1', \ldots, v_n')$ be the received values in the previous step by the workers, instead of the honestly computed valyes $(v_1, \ldots, v_n)$.
   (b) If one of the proofs π_{i1}, π_{i2} or π_{i3} is invalid, the verifier set $b_i = 0$ else it sets $b_i = 1$.
   (c) Since $\Delta(\mathbf{v}', \mathbf{v}) \leq d < (n-t)/2$ from assumption, $\mathcal{V}$ computes $\mathbf{v}$ from $\mathbf{v}'$ by decoding algorithm (e.g. Berlekamp-Welch) for Reed-Solomon codes. Set $\mathsf{C} = \{i \in [n] : v_i \neq v_i' \vee b_i = 0\}$ and let $\mathbf{H}_Q = (h_{jk})$ denote the matrix guaranteed by Lemma 1 for $Q = [n] \backslash \mathsf{C} = \{i_1, \ldots, i_q\}$ for $q \in \mathbb{N}$.
7. **Output using Honest Messages:** $\mathcal{V}$ outputs $(1, \mathsf{C})$ if $\left(\prod_{j \in [q]} A_{i_j}^{h_{jk}}\right)_{k=1,\ldots,n-t} = (z, \mathbf{0}^{n-t-1})$, and $(0, \{\mathcal{P}\})$ otherwise.

Theorem 1. *Assuming that* CSP *satisfies completeness, knowledge-soundness and zero-knowledge with* $O(\log \ell)$*-communication overhead,* Π_{dlog} *is a* $\mathsf{DPoK}_{\mathsf{SSS},\mathsf{DlogGen}}$ *(as per Definition 6) for relation generator* $\mathsf{DlogGen}$ *and* (t, n)*-*SSS *with the following properties:*

- ***Security:*** *t-private and d-robust, for $d < \mathsf{dist}/2$, where $\mathsf{dist} = (n - t)$ is the minimum distance of the Reed-Solomon code induced by (t, n)-SSS.*
- ***Efficiency:*** *$O(n)$ communication over point-to-point channels and $O(n \log \ell)$ communication over broadcast channels.*

We defer the proof of the theorem to the full version of the paper [24].

Generalization to Threshold Linear Secret Sharing. We can generalize the above protocol to work with *any* threshold linear secret sharing (TLSS) scheme. In the generalized version, the corruption threshold for robust completeness depends on the exact distance of the linear code induced by the TLSS scheme. As a corollary, we derive concrete bounds on the corruption threshold for robust completeness when using *replicated secret sharing.* The relevant technical details appear in Appendix C of the full version of the paper [24].

Round Efficient DPoK for Discrete Log. In Appendix E of the full version of the paper [24], we describe a round-efficient version of Π_{dlog} in the random oracle model (obtained using the Fiat-Shamir heuristic), which we call $\Pi^{\mathsf{FS}}_{\mathsf{dlog}}$. We highlight here that, while Π_{dlog} requires a logarithmic (in the size of the witness) number of rounds of interaction, the round-efficient version $\Pi^{\mathsf{FS}}_{\mathsf{dlog}}$ only requires a *constant* number of rounds of interaction. Apart from this, $\Pi^{\mathsf{FS}}_{\mathsf{dlog}}$ satisfies the same robust completeness, knowledge soundness and zero-knowledge properties as Π_{dlog}, albeit in the random oracle model.

4 DPoK for BBS+ Signatures over Secret-Shared Inputs

In this section, we build upon our (publicly verifiable) DPoK for the discrete log relation to design a protocol that allows a prover $\mathcal{P}$ to prove knowledge of a BBS+ (or PS) signature on a secret-shared input. Concretely, suppose that the prover $\mathcal{P}$ holds a BBS+ (or PS) signature σ on a message $\mathbf{m}$ under a public key pk, where $\mathbf{m}$ is secret-shared across n parties $\mathcal{W}_1, \ldots, \mathcal{W}_n$ (i.e. each worker $\mathcal{W}_i$ holds a share $\mathbf{m}_i$). The goal of the protocol is to allow the prover $\mathcal{P}$ to convince a designated verifier $\mathcal{V}$ that σ is a valid signature on $\mathbf{m}$ under pk, *without* revealing σ in the clear (this helps realize the desired property of signature unlinkability, as explained subsequently). We also present similar PoK protocols for PS signatures [44] over secret-shared inputs in Appendix G of the full version of the paper [24]. Looking ahead, we use these protocols as building blocks to design our compiler for upgrading any secret-sharing based MPC protocol into an authenticated version of the same protocol, where the (secret-shared) inputs are authenticated using BBS+(or PS) signatures as above.

We start by defining the relation for BBS+ signature verification.

Definition 7 (BBS+ Relation). *Let* BBSGen *denote the relation generator, such that* $\mathsf{BBSGen}(1^\lambda, \ell)$ *outputs a bilinear group* $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e, p) \leftarrow_R BBS.\mathsf{Setup}(1^\lambda)$*. The corresponding relation* $\mathcal{R}^{\mathrm{bbs}}$ *is defined by* $(\mathbf{x}, (\mathbf{m}, \mathbf{t})) \in \mathcal{R}^{\mathrm{bbs}}$ *for* $\mathbf{x} = \mathsf{pk} = (g_1, w, h_0, \ldots, h_\ell) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1^\ell$, $\mathbf{m} = (m_1, \ldots, m_\ell) \in \mathbb{F}_p^\ell$ *and* $\mathbf{t} = \sigma = (A, \beta, s) \in \mathbb{G}_1 \times \mathbb{F}_p^2$ *if* $e(A, wg_2^\beta) = e(g_1 h_0^s \prod_{i=1}^\ell h_i^{m_i}, g_2)$*.*

Protocol $\Pi_{\mathsf{bbs+}}$

- **Public Key** $\mathsf{pk} = (w, h_0, \ldots, h_\ell)$
- **$\mathcal{P}$'s inputs**: Message $\mathbf{m} = (m_1, \ldots, m_\ell) \in \mathbb{F}_p^\ell$ and signature $\sigma = (A, \beta, s)$ on $\mathbf{m}$, with $A = \left(g_1 h_0^s \prod_{i=1}^\ell h_i^{m_i}\right)^{\frac{1}{\beta+x}}$, such that $(\mathsf{pk}, (\mathbf{m}, \sigma)) \in \mathcal{R}^{\mathrm{bbs}}$
- **$\mathcal{W}_i$'s inputs :** $\mathcal{W}_i$ possesses the i^{th} share $\mathbf{m}_i$ of the message vector $\mathbf{m}$, such that $\mathsf{Reconstruct}(\mathbf{m}_1, \ldots, \mathbf{m}_n) = \mathbf{m}$
- **Pre-processing** : $\mathcal{P}$ samples $u \leftarrow_R \mathbb{F}_p^*, r \leftarrow_R \mathbb{F}_p, \eta \leftarrow_R \mathbb{F}_p$, and computes $d = b^u \cdot h_0^{-r}$ and $t = s - r \cdot v$ where $v = u^{-1}, b = g_1 h_0^s \prod_{i=1}^\ell h_i^{m_i}$. $\mathcal{P}$ computes $(r_1, \ldots, r_n) \leftarrow_R \mathsf{Share}(r)$, $(v_1, \ldots, v_n) \leftarrow_R \mathsf{Share}(v)$, $(\beta_1, \ldots, \beta_n) \leftarrow_R \mathsf{Share}(\beta)$, $(t_1, \ldots, t_n) \leftarrow_R \mathsf{Share}(t)$, $(\eta_1, \ldots, \eta_n) \leftarrow_R \mathsf{Share}(\eta)$. $\mathcal{P}$ sends the shares $(r_i, v_i, \beta_i, t_i, \eta_i)$ to $\mathcal{W}_i$, for all $i \in [n]$.
  In other words, each $\mathcal{W}_i$ locally holds the i-th share $\mathbf{s}_i = (\mathbf{m}_i, r_i, v_i, \beta_i, t_i, \eta_i)$ such that
  $$\mathbf{s} = (\mathbf{m}, r, v, \beta, t) = \mathsf{Reconstruct}\left(\{\mathbf{s}_i\}_{i \in [n]}\right).$$
- **Interactive Protocol:**
  1. $\mathcal{P}$ computes $A' = A^u$, $\bar{A} = (A')^{-\beta} \cdot b^u (= (A')^x)$, where $b = g_1 h_0^s \prod_{i=1}^\ell h_i^{m_i}$ and $d = b^u \cdot h_0^{-r}$. $\mathcal{P}$ sets $C = d^{-v} h_0^{t-\eta}$, $D = h_0^\eta \prod_{i=1}^\ell h_i^{m_i}$, and broadcasts $(A', \bar{A}, d, C, D)$ to each $\mathcal{W}_i$ and $\mathcal{V}$.
  2. The workers $\mathcal{W}_i$, $i \in [n]$ and $\mathcal{V}$ run the DPoK Π_{dlog} for the relation $D = h_0^\eta \prod_{i=1}^\ell h_i^{m_i}$, where $(\eta, m_1, \ldots, m_\ell)$ are secret-shared across the workers; and $\mathbf{g} = (h_0, \ldots, h_\ell)$, $z = D$ is available to all parties.
  3. Simultaneously, the workers $\mathcal{W}_i$, $i \in [n]$ and $\mathcal{V}$ run the DPoK Π_{dlog} for the relation $C = d^{-v} h_0^{t-\eta} \wedge \frac{\bar{A}}{d} = (A')^{-\beta} h_0^r$, where (v, η) and (β, r) are secret-shared; and $\mathbf{g} = ((d, h_0), (A', h_0))$, $z = (C, \frac{\bar{A}}{d})$ is available to all parties.
  4. $\mathcal{V}$ accepts if $C \cdot D = g_1^{-1}$, *and* $e(A', w) = e(\bar{A}, g_2)$, *and both* instances of Π_{dlog} accept.

Our DPoK Protocol $\Pi_{\mathsf{bbs+}}$. We build upon the robust complete DPoK Π_{dlog} for discrete log to propose a DPoK achieving robust completeness for BBS+ signatures, which allows a designated prover $\mathcal{P}$, to show knowledge of a BBS+ signature (A, β, s) over the message $\mathbf{m} \in \mathbb{F}_p^\ell$ that is secret-shared amongst the workers $\mathcal{W}_1, \ldots, \mathcal{W}_n$. Recall that this PoK involved the following steps: (i) the prover randomly chooses some auxiliary inputs, and combines them with the signature to output a randomized first message (this randomization ensures unlinkability), and then (ii) the prover shows knowledge of these auxiliary inputs and components of the signature satisfying discrete-log relations determined by the first message.

Our BBS+ DPoK over secret-shared inputs follows a similar blueprint, where the prover similarly randomizes the first message using certain auxiliary inputs. In our case, the prover: (i) secret-shares the auxiliary inputs to the workers using point-to-point channels (this step is unique to our protocol and is designed to facilitate distributed proving in the subsequent steps), and (ii) broadcasts the first message to the workers *and* the verifier (this step uses broadcast channels and is conceptually similar to the PoK over non-distributed inputs). At this point, the problem reduces to a DPoK for the discrete log relation. We handle this using our robust complete DPoK Π_{dlog} for discrete log.

We prove the $\Pi_{\mathsf{bbs+}}$ to be a DPoK for the relation generator BBSGen in the following theorem.

Theorem 2. *Assuming that* Π_{dlog} *is a* $\mathsf{DPoK}_{\mathsf{SSS,DlogGen}}$ *for relation generator* $\mathsf{DlogGen}$ *and* (t,n)-SSS*,* $\Pi_{\mathsf{bbs+}}$ *is a* DPoK *for the relation generator* BBSGen *and* (t,n)-SSS *with:*

- ***Security****: t-private and d-robust, for* $d < \mathsf{dist}/2$*, where* $\mathsf{dist} = (n-t)$ *is the minimum distance of the Reed-Solomon code induced by* (t,n)-SSS*.*
- ***Efficiency****:* $O(n)$ *communication over point-to-point channels and* $O(n \log \ell)$ *communication over broadcast channels.*

We defer the proof to the full version of the paper [24].

Protocol $\Pi_{\mathsf{bbs\text{-}auth\text{-}opt}}$

- **Public Parameters**: $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e, p) \leftarrow_R \mathsf{BBSGen}(1^\lambda)$ defining BBS+ relation $\mathcal{R}^{\mathrm{bbs}}$. Let $\mathsf{pk} = (g_1, w = g_2^x, h_0, \ldots, h_\ell)$ be a known public key for secret key $\mathsf{sk} = x \leftarrow_R \mathbb{F}_p$.
- P_i**'s inputs**:
  - Message $\mathbf{m}_i \in \mathbb{F}_p^\ell$ and signature $\sigma_i = (A_i, \beta_i, s_i)$ on $\mathbf{m}_i$ under pk.
  - i^{th} share of the message $\mathbf{m}_j$ of P_j.
- **Pre-processing**: $\mathcal{P}_i$ samples $u_i \leftarrow_R \mathbb{F}_p^*, r_i \leftarrow_R \mathbb{F}_p, \eta_i \leftarrow_R \mathbb{F}_p$, and computes $d_i = b_i^{u_i} \cdot h_0^{-r_i}$ and $t_i = s_i - r_i \cdot v_i$ where $v_i = u_i^{-1}, b_i = g_1 h_0^{s_i} \prod_{i=1}^{\ell} h_i^{m_i}$. and secret shares $r_i, v_i, t_i, \eta_i, \beta_i$ among $P_1, \ldots, P_n$. All parties set $\mathbf{g} = (h_0, \ldots, h_\ell)$.
- **Interactive Protocol**
  1. $\mathcal{P}_i$, $i \in [n]$ computes $A_i' = A_i^{u_i}$, $\bar{A}_i = (A_i')^{-\beta} \cdot b_i^u (= (A_i')^x)$. $\mathcal{P}$ sets $C_i = d_i^{-v_i} h_0^{t_i - \eta_i}$, $D_i = \mathbf{g}^{\eta_i, \mathbf{m}_i}$, and broadcasts $(A_i', \bar{A}_i, d_i, C_i, D_i)$.
  2. The verifier samples a challenge $\boldsymbol{\gamma} \leftarrow_R \mathbb{F}_p^\ell$ and broacasts it. Each P_i then computes $\mathbf{y}_i = \sum_{j \in [n]} \gamma^j (\eta_{ij}, \mathbf{m}_{ij})$, where $\eta_{ij}, \mathbf{m}_{ij}$ denotes $\mathcal{P}_i$'s share of $\mathcal{P}_j$'s inputs $\mathbf{m}_j, \eta_{ij}$.
  3. All parties compute $D = \prod_{j \in [n]} D_j^{\gamma^j}$.

     Parties hold shares $\mathbf{y}_i$ of $\mathbf{y}$ satisfying $\mathbf{g}^{\mathbf{y}} = D$

  4. Parties run the interactive phase of the protocol $\Pi_{\mathsf{nr\text{-}dlog}}$ on statement D with $\mathbf{g}$ as the generator. They run the interactive phase of the protocol

$\Pi_{\mathsf{nr\text{-}dlog}}$ on statements $C_i = d_i^{-v_i} h_0^{t_i - \eta_i} \wedge \frac{\bar{A}_i}{d_i} = (A_i')^{-\beta_i} h_0^{r_i}$, for each $i \in [n]$ with generators (d_i, h_0) and (A_i', h_0) respectively.

5. Parties also check that $e\left(\prod_{i=1}^{n} A_i', w\right) = e\left(\prod_{i=1}^{n} \bar{A}_i, g_2\right)$ holds.

– **Output**: P_j outputs $b_j = 1$ if all the above protocols lead to accept.

Efficiently Batching BBS+ PoKs. We now present the protocol $\Pi_{\mathsf{bbs\text{-}auth\text{-}opt}}$ which efficiently batches n parallel instances of the protocol $\Pi_{\mathsf{bbs+}}$ with the party $\mathcal{P}_i$ acting as the prover in the i^{th} instance of the protocol. The optimization exploits the fact that each party needs to prove a linear (in exponents) relation over large part of its witness (the message vector), which can be reduced via a random challenge to proving a linear relation over the linearly combined messages. However we lose robustness: we can no longer identify the corrupt parties or a corrupt prover using error-correction as in $\Pi_{\mathsf{bbs+}}$, as the combined witness cannot be attributed to a specific party. Thus, we simply abort if one of the checks in the underlying protocol $\Pi_{\mathsf{nr\text{-}dlog}}$ fails.

Round Efficient DPoK for BBS+ Signatures. Finally, note that by replacing Π_{dlog} with its round efficient version $\Pi_{\mathsf{dlog}}^{\mathsf{FS}}$ in the random oracle model (obtained using the Fiat-Shamir heuristic, presented in Appendix E of the full version of the paper [24]) in steps (2) and (3) of the Interactive Phase, we obtain a round efficient version of the protocol, which we call $\Pi_{\mathsf{bbs+}}^{\mathsf{FS}}$. Observe that $\Pi_{\mathsf{bbs+}}^{\mathsf{FS}}$ requires constant rounds of interaction, as compared to logarithmic (in the size of the message) rounds of interaction for $\Pi_{\mathsf{bbs+}}$, and satisfies the same robust completeness, knowledge soundness and zero-knowledge properties as $\Pi_{\mathsf{bbs+}}$, albeit in the random oracle model.

5 Compiler for Authenticated MPC

In this section we present our compiler for MPC with input authentication that outputs an MPC protocol where each input is authenticated using a BBS+ signature under a common (public) verification key. In Appendix G of the full version of the paper [24], we outline a similar compiler based on PS signatures.

Class of MPC Protocols. Our compiler takes advantage of the observation that a large class of secret-sharing based MPC protocols share the following template. (i) There is an input sharing phase where parties secret-share their inputs, and (ii) when using secret sharing schemes with certain thresholds ($t_{\mathsf{sh}} < |H|$), the input of parties is completely determined at the end of the input sharing phase. This means that using inputs inconsistent with this sharing is considered deviating, against which the protocol is secure. This is precisely where our compiler performs well: verification of authenticity (or any other predicate) on the inputs can be done fully outside the MPC by running a DPoK on the shares. (iii) For an MPC protocol of this template, there exists a simulator

$\mathsf{Sim} = (\mathsf{Sim}_{\mathsf{sh}}, \mathsf{Sim}_{\mathsf{on}})$, where $\mathsf{Sim}_{\mathsf{sh}}$ deterministically extracts the inputs of corrupt parties, and $\mathsf{Sim}_{\mathsf{on}}$ simulates the protocol view.

Features of Our Compiler. Our compiler allows identification of all (malicious) parties with non-authenticated inputs (this is a consequence of the robust completeness property of Π_{dlog} used inside $\Pi_{\mathsf{bbs+}}$). We further note that our robust protocol Π_{dlog} tolerates a maximum corruption threshold of $t < n/3$ (assuming that the secret-sharing used is Shamir's secret sharing). Hence, our compiled MPC protocol also tolerates a maximum corruption threshold of $t < n/3$. Using the non-robust version will result in a non-robust compiler that retains the $t < n/2$ threshold of the underlying MPC.

The Desired Ideal Functionality. We define below the desired ideal functionality $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{authid}}$ for MPC with input authentication.

Functionality $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{auth}}$

Inputs

The ideal functionality receives from each party P_i an input-signature pair of the form $(\mathbf{x}_i, \sigma_i)$ under the public verification key pk.

Verify Authenticity

1. If $\mathsf{Ver}(\mathsf{pk}, x_i, \sigma_i) \neq 1$ for some party P_i, then output a set of corrupted parties C and abort.
2. Otherwise, proceed to computation.

Computation

Invoke the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$ for Π_{mpc} on inputs $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$.

5.1 Our Compiler

We now present a formal description of our compiler. Let $\Pi_{\mathsf{mpc}} = (\Pi_{\mathsf{sh}}, \Pi_{\mathsf{on}})$ be a secret-sharing based MPC protocol that guarantees security with abort against malicious corruptions of a dishonest majority of the parties $\{P_1, \ldots, P_n\}$, where:

– Π_{sh} denotes the secret-sharing phase of Π_{mpc} and consists of the steps used by each party P_i for $i \in [n]$ to secret-share its input $\mathbf{x}_i \in \mathbb{F}_p^\ell$ to all of the other parties (throughout, we assume that this sharing is done using a linear secret-sharing scheme $(\mathsf{Share}, \mathsf{Reconstruct})$.
– Π_{on} denotes the remaining steps of the protocol Π_{mpc} where the parties interact to compute $y = f(\mathbf{x}_1, \ldots, \mathbf{x}_n)$.

Protocol $\Pi_{\mathsf{ampc}} = (\overline{\Pi}_{\mathsf{sh}}, \overline{\Pi}_{\mathsf{on}})$

- **Inputs:** All parties hold public parameters and the verification key pk of a BBS+ signature scheme. Party P_i has input $\mathbf{x}_i \in \mathbb{F}_p^{\ell}$, together with a signature σ_i, such that $(\mathsf{pk}, (\mathbf{x}_i, \sigma_i)) \in \mathcal{R}^{\mathrm{bbs}}$.
- $\overline{\Pi}_{\mathsf{sh}}$: This phase is identical to Π_{sh}, i.e., each party P_i shares its input $\mathbf{x}_i$ to all other parties exactly as in Π_{sh}.
- $\overline{\Pi}_{\mathsf{on}}$: In this phase, the parties do the following:
  - For each $j = 1, \ldots, n$, the parties execute an instance of $\Pi_{\mathsf{bbs+}}$ for $(\mathsf{pk}, (\mathbf{x}_j, \sigma_j)) \in \mathcal{R}^{\mathrm{bbs}}$ with $\mathcal{P}_j$ acting as the Prover, $\mathcal{P}_1, \ldots, \mathcal{P}_n$ constituting the workers and $\mathcal{P}_i$, $i \neq j$ acting as verifiers, .
    If any party outputs 0 at the end of this phase, the protocol aborts.
  - Otherwise, the parties jointly execute Π_{on}.

In the description of our compiler, we assume that each party P_i holds a BBS+ signature σ_i on its input $\mathbf{x}_i$ with respect to a common public verification key pk. The compiler runs n instances of $\Pi_{\mathsf{bbs+}}$, where for instance i, party P_i acts as the prover and all other parties P_j for $j \neq i$ act as verifiers. Given $\Pi_{\mathsf{mpc}} = (\Pi_{\mathsf{sh}}, \Pi_{\mathsf{on}})$, our robust compiler outputs an authenticated MPC protocol $\Pi_{\mathsf{ampc}} = (\overline{\Pi}_{\mathsf{sh}}, \overline{\Pi}_{\mathsf{on}})$. The compiler Π_{ampc} is described above.

Theorem 3 (Security of Π_{ampc}). *Assuming that: (a) the MPC protocol* Π_{mpc} *securely emulates the ideal functionality* $\mathcal{F}_{\mathsf{MPC}}$*, and (b)* Π_{dlog} *is a* $\mathsf{DPoK}_{\mathsf{SSS},\mathsf{DlogGen}}$ *for relation generator* $\mathsf{DlogGen}$ *and* (t, n)-SSS *our compiled MPC protocol with input authentication* Π_{ampc} *securely emulates the ideal functionality* $\mathcal{F}_{\mathsf{MPC}}^{\mathsf{auth}}$ *for the same corruption threshold of* $t < n/3$.

We defer the proof of this theorem to the full version of the paper [24].

Round Efficient Compiler for Authenticated MPC. Finally, it is easy to see that invoking the round efficient DPoK $\Pi_{\mathsf{bbs+}}^{\mathsf{FS}}$ protocol instead of the DPoK $\Pi_{\mathsf{bbs+}}$ protocol enables us to obtain a round efficient version of our compiler. The round efficient version achieves the same security guarantees as the compiler presented above, albeit in the random oracle model.

6 Implementation and Evaluation

In this section, we present a prototype implementation of our compiler using $\Pi_{\mathsf{bbs\text{-}auth\text{-}opt}}$ for BBS+ signatures. We test and benchmark our implementation on a 16GB system with Intel Core i5-9400 CPU clocked at 2.9GHz and running Ubuntu Linux 20.04. All the benchmarks use single execution thread. We use the implementation of `BN128` elliptic curve from the library `libff` [48] to implement $\Pi_{\mathsf{bbs\text{-}auth\text{-}opt}}$ with $(t, 2t+1)$-Shamir secret sharing[9]. We then integrate

[9] We do not implement broadcast functionality cryptographically. To obtain the benchmarks we implement a server acting as a broadcast hub. Efficient broadcast can be implemented for our setting based on [30].

our implementation of $\Pi_{\mathsf{bbs\text{-}auth\text{-}opt}}$ with a maliciously secure implementation of Shamir-secret sharing-based MPC from the well-known `MP-SPDZ` library [36] to obtain an implementation of authenticated MPC[10].

Table 2. Benchmarks for the secure KPI application with 3 and 5 parties by comparing our DPoK-based approach for MPC input authentication with the naïve approach of validating BBS+ signatures inside MPC (which involves computing MiMC hashes inside MPC). The second column titled "Rows" indicates the number of rows in each party's dataset (the number of columns is fixed to 10).

# Parties	# Rows	Vanilla MPC		DPoK Overhead	
		Comm(MB)	Time (s)	Comm.(KB)	Time (s)
3	100	1733	6.67	13	0.519
	1000	16754	64	15	18
	2000	33398	129	15.3	65
	4000	66502	260	15.8	246
5	100	8838	26	28	0.643
	1000	87747	265	31	20
	2000	175671	521	32	76
	4000	350658	958	33	312

Evaluation and Discussion. We benchmark both $\Pi_{\mathsf{bbs\text{-}auth\text{-}opt}}$ (in a standalone manner) and the final authenticated MPC protocol (obtained by integrating $\Pi_{\mathsf{bbs\text{-}auth\text{-}opt}}$ with MP-SPDZ [36] as specified in our compiler) in the setting of the industry KPI application outlined in the introduction. We consider two instances of the KPI application, with 3 and 5 parties, where each party's dataset has 10 columns and variable number of rows (between 100 and 4000). We summarize the overheads for vanilla unauthenticated computation using MP-SPDZ, as well as the additional overheads incurred by the compiled authenticated MPC, in Table 2. It is readily apparent that the communication overhead of input authentication over vanilla MPC are minimal. The computational overhead grows with input size, which is unavoidable to an extent, as BBS+ signature verification involves algebraic operations that grow with the size of the input. The major contributor to the computational overheads are the instances of NIPK, which may be parallelized for large input sizes. We leave such optimized implementations as interesting future work.

[10] An anonymized version of our code repository is available here: https://anonymous.4open.science/r/authenticatedMPC-476E/CMakeLists.txt.

References

1. Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.
2. Diego F. Aranha, Anders P. K. Dalskov, Daniel Escudero, and Claudio Orlandi. Improved threshold signatures, proactive secret sharing, and input certification from LSS isomorphisms. In Patrick Longa and Carla Ràfols, editors, *LATINCRYPT 2021*, volume 12912, pages 382–404, 2021.
3. Thomas Attema and Ronald Cramer. Compressed Σ-protocol theory and practical application to plug & play secure algorithmics. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 513–543. Springer, Cham, August 2020.
4. Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-TAA. In Roberto De Prisco and Moti Yung, editors, *SCN 06*, volume 4116 of *LNCS*, pages 111–125. Springer, Berlin, Heidelberg, September 2006.
5. Carsten Baum. On garbling schemes with and without privacy. In Vassilis Zikas and Roberto De Prisco, editors, *SCN 16*, volume 9841 of *LNCS*, pages 468–485. Springer, Cham, August / September 2016.
6. Carsten Baum, Robin Jadoul, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. Feta: Efficient threshold designated-verifier zero-knowledge proofs. Cryptology ePrint Archive, Paper 2022/082, 2022. https://eprint.iacr.org/2022/082.
7. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.
8. Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Cham, May 2019.
9. Fabrice Benhamouda, Tancrède Lepoint, Julian Loss, Michele Orrù, and Mariana Raykova. On the (in)security of ROS. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 33–53. Springer, Cham, October 2021.
10. Marina Blanton and Fattaneh Bayatbabolghani. Efficient server-aided secure two-party function evaluation with applications to genomic computation. *PoPETs*, 2016(4):144–164, October 2016.
11. Marina Blanton and Myoungin Jeong. Improved signature schemes for secure multi-party computation with certified inputs. In Javier López, Jianying Zhou, and Miguel Soriano, editors, *ESORICS 2018, Part II*, volume 11099 of *LNCS*, pages 438–460. Springer, Cham, September 2018.
12. Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 41–55. Springer, Berlin, Heidelberg, August 2004.
13. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, Cham, August 2019.
14. Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log

setting. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, Berlin, Heidelberg, May 2016.
15. Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong diffie hellman assumption revisited. In *TRUST 2016*, volume 9824, pages 1–20. Springer, 2016.
16. Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 93–118. Springer, Berlin, Heidelberg, May 2001.
17. Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 21–30. ACM Press, November 2002.
18. Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Cham, May 2020.
19. Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI 2017*, pages 259–282. USENIX Association, 2017.
20. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Berlin, Heidelberg, September 2013.
21. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Advances in Cryptology - CRYPTO*, pages 572–590, 2007.
22. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Berlin, Heidelberg, August 2012.
23. Pankaj Dayama, Arpita Patra, Protik Paul, Nitin Singh, and Dhinakaran Vinayagamurthy. How to prove any NP statement jointly? efficient distributed-prover zero-knowledge protocols. *Proc. Priv. Enhancing Technol.*, 2022(2):517–556, 2022.
24. Moumita Dutta, Chaya Ganesh, Sikhar Patranabis, and Nitin Singh. Compute, but verify: Efficient multiparty computation over authenticated inputs. Cryptology ePrint Archive, Report 2022/1648, 2022.
25. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Berlin, Heidelberg, August 1987.
26. Dario Fiore and Anca Nitulescu. On the insecurity of snarks in the presence of oracles. In *Proceedings, Part I, of the 14th International Conference on Theory of Cryptography - Volume 9985*, page 108-138, Berlin, Heidelberg, 2016. Springer-Verlag.
27. Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019.
28. Chaya Ganesh, Hamidreza Khoshakhlagh, Markulf Kohlweiss, Anca Nitulescu, and Michal Zajac. What makes fiat–shamir zksnarks (updatable srs) simulation

extractable? Cryptology ePrint Archive, Paper 2021/511, 2021. https://eprint.iacr.org/2021/511.
29. Chaya Ganesh, Claudio Orlandi, Mahak Pancholi, Akira Takahashi, and Daniel Tschudi. Fiat-shamir bulletproofs are non-malleable (in the random oracle model). Cryptology ePrint Archive, Paper 2023/147, 2023. https://eprint.iacr.org/2023/147.
30. Chaya Ganesh and Arpita Patra. Broadcast extensions with optimal communication and round complexity. In George Giakkoupis, editor, *35th ACM PODC*, pages 371–380. ACM, July 2016.
31. Sanjam Garg, Aarushi Goel, Abhishek Jain, Guru-Vamsi Policharla, and Sruthi Sekar. zkSaaS: Zero-Knowledge SNARKs as a service. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4427–4444, Anaheim, CA, August 2023. USENIX Association.
32. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.
33. Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Berlin, Heidelberg, May 2016.
34. Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, and Mor Weiss. Your reputation's safe with me: Framing-free distributed zero-knowledge proofs. Cryptology ePrint Archive, Paper 2022/1523, 2022. https://eprint.iacr.org/2022/1523.
35. Jonathan Katz, Alex J. Malozemoff, and Xiao Wang. Efficiently enforcing input validity in secure two-party computation. Cryptology ePrint Archive, Report 2016/184, 2016. https://ia.cr/2016/184.
36. Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020.
37. Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 549–560. ACM Press, November 2013.
38. Joe Kilian. Founding cryptography on oblivious transfer. In *20th ACM STOC*, pages 20–31. ACM Press, May 1988.
39. Tobias Looker, Vasilis Kalos, Andrew Whitehead, and Mike Lodder. The bbs signature scheme. Internet Engineering Task Force, 2022. https://identity.foundation/bbs-signature/draft-irtf-cfrg-bbs-signatures.html.
40. Alex Ozdemir and Dan Boneh. Experimenting with collaborative zk-SNARKs: Zero-knowledge proofs for distributed secrets. Cryptology ePrint Archive, Report 2021/1530, 2021.
41. Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Berlin, Heidelberg, August 1992.
42. Torben Pryds Pedersen. Distributed provers with applications to undeniable signatures. In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 221–242. Springer, Berlin, Heidelberg, April 1991.
43. Chris Peikert. On error correction in the exponent. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 167–183. Springer, Berlin, Heidelberg, March 2006.

44. David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 111–126. Springer, Cham, February / March 2016.
45. Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, New York, August 1990.
46. Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991.
47. Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *ACNS 16International Conference on Applied Cryptography and Network Security*, volume 9696 of *LNCS*, pages 346–366. Springer, Cham, June 2016.
48. MIT SCIPR Lab. libff: C++ library for finite fields and elliptic curves. https://github.com/scipr-lab/libff, 2023. https://github.com/scipr-lab/libff.
49. Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.
50. Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 675–692. USENIX Association, August 2018.
51. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.
52. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.
53. Yihua Zhang, Marina Blanton, and Fattaneh Bayatbabolghani. Enforcing input correctness via certification in garbled circuit evaluation. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *ESORICS 2017, Part II*, volume 10493 of *LNCS*, pages 552–569. Springer, Cham, September 2017.

Dishonest Majority Constant-Round MPC with Linear Communication from DDH

Vipul Goyal[1,2](✉), Junru Li[3], Ankit Kumar Misra[4], Rafail Ostrovsky[4], Yifan Song[3,5], and Chenkai Weng[6]

[1] NTT Research, Sunnyvale, USA
[2] Carnegie Mellon University, Pittsburgh, USA
vipul@cmu.edu
[3] Tsinghua University, Beijing, China
jr-li24@mails.tsinghua.edu.cn, yfsong@mail.tsinghua.edu.cn
[4] UCLA, Los Angeles, USA
ankitkmisra@g.ucla.edu, rafail@cs.ucla.edu
[5] Shanghai Qi Zhi Institute, Shanghai, China
[6] Arizona State University, Tempe, USA
Chenkai.Weng@asu.edu

Abstract. In this work, we study constant round multiparty computation (MPC) for Boolean circuits against a fully malicious adversary who may control up to $n-1$ out of n parties. Without relying on fully homomorphic encryption (FHE), the best-known results in this setting are achieved by Wang et al. (CCS 2017) and Hazay et al. (ASIACRYPT 2017) based on garbled circuits, which require a quadratic communication in the number of parties $O(|C| \cdot n^2)$. In contrast, for non-constant round MPC, the recent result by Rachuri and Scholl (CRYPTO 2022) has achieved linear communication $O(|C| \cdot n)$.

In this work, we present the first concretely efficient constant round MPC protocol in this setting with linear communication in the number of parties $O(|C| \cdot n)$. Our construction can be based on any public-key encryption scheme that is linearly homomorphic for public keys. Our work gives a concrete instantiation from a variant of the El-Gamal Encryption Scheme assuming the DDH assumption. The analysis shows that when the computational security parameter $\lambda = 128$ and statistical security parameter $\kappa = 80$, our protocol achieves a smaller communication than Wang et al. (CCS 2017) when there are 16 parties for AES circuit and 8 parties for general Boolean circuits (where we assume that the numbers of AND gates and XOR gates are the same). When comparing with the recent work by Beck et al. (CCS 2023) that achieves constant communication complexity $O(|C|)$ in the strong honest majority setting ($t < (1/2 - \epsilon)n$ where ϵ is a constant), our protocol is better as long as $n < 3500$ (when $t = n/4$ for their work).

1 Introduction

Secure multiparty computation (MPC) [8,13,21,35,43] allows a set of n parties to jointly compute a public function on their private inputs. The efficiency of

K.-M. Chung and Y. Sasaki (Eds.): ASIACRYPT 2024, LNCS 15489, pp. 167–199, 2025.
https://doi.org/10.1007/978-981-96-0938-3_6

MPC protocols can be measured from various aspects, and the most two common criteria are the communication complexity and the round complexity.

Communication-Efficient but Non-Constant Round MPC. The well-known SPDZ protocol was first introduced by Damgård et al. [17], which achieves a very efficient online protocol whose communication complexity grows linearly with the number of parties in the dishonest majority setting. Due to its potential of being used in practice, a long line of works [1,7,16,19,26–28] focus on improving both the offline phase and the online phase of the SPDZ protocol. Thanks to the recent progress of pseudo-random correlation generators (PCG) [9,10,41], Rachuri and Scholl [36] have achieved a linear communication complexity $O(|C|n)$ in both the offline phase and the online phase.

However, all SPDZ-style protocols suffer a large round complexity that grows linearly with the depth of the circuit. For circuits with large depths in the WAN setting, the network latency may become the main bottleneck.

Constant Round but Communication-Heavy MPC. Due to the round complexity of SPDZ protocols, another line of works target constant round MPC. Without relying on fully homomorphic encryption (which is not considered to be efficient in practice), the most common way is to follow the BMR template [2] that generalizes the Yao's garbled circuits [44] from the two-party setting to the multiparty setting. Despite the significant progress in [5,6,29], [4,23–25,39,42], the best-known result in the dishonest majority setting still requires a quadratic communication complexity in the number of parties $O(|C|n^2)$, which is insufficient to support applications that involve hundreds or thousands of parties.

The efficiency gap between these two types of MPC protocols leads to our following question:

"Can we construct a fully malicious MPC protocol in the dishonest majority setting that achieves the best in both worlds, i.e., with constant rounds and linear communication complexity?"

1.1 Our Contributions

In this work, we answer the above question affirmatively by presenting the first concretely efficient constant-round and fully malicious MPC protocol with linear communication in the dishonest majority setting $(t < n)$.

Theorem 1. *Assuming DDH, LPN, and random oracles, there is a computationally secure constant-round MPC protocol against a fully malicious adversary controlling up to $n-1$ parties with communication of $O(|C|n\lambda)$ bits, where λ is the computational security parameter.*

Our construction has the following features.

– **Communication Complexity.** To compute a circuit C of size $|C|$, the total communication complexity of our protocol is $O(|C|n\lambda)$ bits, where λ is the computational security parameter.

- **Assumptions.** Our protocol makes a black-box use of building blocks in Le Mans [36], which can be instantiated based on the LPN assumption. Beyond the building blocks in Le Mans [36], the garbling phase of our construction only requires the DDH assumption and symmetric-key cryptographic assumptions (random oracle or pseudo-random generator).
- **Concrete Efficiency.** Comparing with the previously best-known results [39,42] with quadratic communication, our protocol achieves a smaller communication as long as there are 16 parties for the AES-128 circuit when the computational security parameter $\lambda = 128$ and statistical security parameter $\kappa = 80$. We note that the work [4] can potentially achieve a linear communication when its underlying SPDZ preprocessing is instantiated by Le Mans [36]. Even after optimization, the communication cost of our protocol outperforms theirs by $11.7\times$ on the AES-128 circuit and $11.3\times$ on the SHA-256 circuit.

Our construction is conceptually simple. We note that the main difficulty in all previous works following the BMR template is to emulate the encryption algorithm of some symmetric-key encryption scheme where both the key k and the message m are secretly shared.

Our first idea is to replace the symmetric-key encryption scheme used in the BMR template by a public-key encryption scheme. In this way, while we still need to keep the private key secret, the public key can be learnt by all parties. Now when emulating the encryption algorithm, only the message to be encrypted is secretly shared.

To allow all parties efficiently generate public-private key pairs (pk, k) where pk is known to all parties while k is (additively) shared among all parties, we make use of a public-key encryption scheme that is linearly homomorphic for public keys: For two key pairs $(\mathsf{pk}_1, k_1), (\mathsf{pk}_2, k_2)$, we require that there is a homomorphic operation $\tilde{+}$ such that $(\mathsf{pk}_1 \tilde{+} \mathsf{pk}_2, k_1 + k_2)$ is also a valid key pair. We show how key pairs can be efficiently generated relying on this property.

Although pk is known to all parties, emulating the encryption algorithm to encrypt a shared message m may still be inefficient. Our second idea is to let each party P_i just encrypt his message share. To be more concrete, suppose m is additively shared to all parties and m_i is held by P_i. We simply let each P_i encrypt m_i by pk and denote the cipher-text by ct_i. Note that a party having the private key k and all cipher-texts $(\mathsf{ct}_1, \ldots, \mathsf{ct}_n)$ can decrypt each m_i and compute m. This allows us to make use of the public-key encryption scheme in a black-box way. We show that this is sufficient for us to achieve a linear communication complexity.

2 Technical Overview and Related Works

We give a high-level overview of the main techniques used in this paper. Our goal is to construct a *constant-round* MPC protocol for a general Boolean circuit C consisting of AND and XOR gates. We focus on the dishonest majority setting, where up to $t = n - 1$ parties can be corrupted.

2.1 Background: Yao's Garbled Circuit and BMR Template

Most of the constant-round MPC protocols in the dishonest majority setting are based on multiparty garbling techniques derived from the well-known BMR protocol given by Beaver, Micali, and Rogaway [2]. At a high level, the BMR technique is to let all parties jointly compute a Yao's garbled circuit of C. We first give a brief review of Yao's garbled circuits.

Review of Yao's Garbled Circuits. Yao's garbled circuit [43] was designed in the two-party setting, where one party acts as the garbler to construct a garbled circuit, and the other party acts as the evaluator to evaluate this garbled circuit such that the evaluator only learns the circuit output and nothing else.

To garble a Boolean circuit C, the garbler first prepares a random bit value λ_w and a pair of labels $(k_{w,0}, k_{w,1})$ for each wire w in the circuit. During the evaluation phase, we want to maintain the invariant that the evaluator learns only $v_w \oplus \lambda_w$ and the corresponding label $k_{w,v_w \oplus \lambda_w}$, where v_w is the actual wire value, protected by the random bit λ_w. To this end, for a gate in C with input wires a, b and output wire c, we want the evaluator to be able to learn $(v_c \oplus \lambda_c, k_{c,v_c \oplus \lambda_c})$ if he holds $(v_a \oplus \lambda_a, k_{a,v_a \oplus \lambda_a})$ and $(v_b \oplus \lambda_b, k_{b,v_b \oplus \lambda_b})$. This can be done by preparing the following 4 ciphertexts. At a high level, we simply use the two labels, one from each input wire, as secret keys to encrypt the proper label for the output wire:

1. Let $f : \{0,1\}^2 \to \{0,1\}$ be the function computed by this gate, which is either the AND function or the XOR function. Let $g : \{0,1\}^2 \to \{0,1\}$ be defined by $g(x, y) = f(x \oplus \lambda_a, y \oplus \lambda_b) \oplus \lambda_c$. Then we have $g(v_a \oplus \lambda_a, v_b \oplus \lambda_b) = v_c \oplus \lambda_c$. We set $\chi_{i,j} = g(i, j)$.
2. The ciphertexts are the following: $\{\mathsf{Enc}_{k_{a,i},k_{b,j}}(\chi_{i,j}, k_{c,\chi_{i,j}})\}_{i,j \in \{0,1\}}$. Then, the evaluator can decrypt the ciphertext with index $(i, j) = (v_a \oplus \lambda_a, v_b \oplus \lambda_v)$ and learn $(v_c \oplus \lambda_c, k_{c,v_c \oplus \lambda_c})$.

Finally, to let the evaluator obtain the function output, the garbler simply sends λ_w associated with the output wires to the evaluator.

The security follows from the fact that the evaluator only learns one of the two labels for each wire. This only allows him to decrypt one of the 4 ciphertexts for each gate. It is important to note that in the 2-party setting, either the garbler is corrupted or the evaluator is corrupted.

BMR Template. In the multiparty setting, we cannot let a single party act as the garbler and let all other parties act as evaluators, since the garbler may collude with some evaluator, and then security would no longer hold. The idea of the BMR construction is to let all parties jointly emulate the garbler. Note that after preparing $(\lambda_w, k_{w,0}, k_{w,1})$ for each wire, all ciphertexts can be computed in parallel. Thus, the computation task of the garbler can be represented by a constant-depth circuit, with size growing linearly in $|C|$. We may use a generic dishonest majority MPC protocol to emulate the garbler within constant rounds.

After all parties securely generate Yao's garbled circuit, each party can act as an evaluator to obtain its function output locally.

When using the state-of-the-art SPDZ-style protocol (Le Mans [36]) to instantiate the generic dishonest majority MPC protocol, it's possible to achieve $O(|C|n)$ communication with constant rounds to emulate the computation task of the garbler. However, an efficiency bottleneck of BMR constructions is that all parties have to emulate the underlying encryption algorithm, which needs to use the underlying symmetric-key encryption scheme in a non-black-box way. Following up works have tried to improve the concrete efficiency of the BMR construction by either making the underlying cryptographic tools used in a black-box way or using concrete instantiations for the symmetric-key encryption scheme. As we will discuss in Sect. 2.5, these works either require a quadratic communication in the number of parties or introduce a large multiplicative overhead.

2.2 Our Solution

Starting Point: Using Public-Key Encryption Schemes. Recall that for every wire w, we need to prepare a pair of labels $(k_{w,0}, k_{w,1})$. These labels are used as secret keys to compute proper ciphertexts. When computing the garbled circuit in a distributed way, all parties only hold shares of secret keys (wire labels) and the messages they want to encrypt. Indeed, emulating the encryption algorithm from shares of the secret key and the message is the main difficulty.

Our starting point is to replace the symmetric-key encryption scheme in the BMR template with a public-key encryption scheme. Then, each wire label becomes a key pair (pk, k) where pk is the public key and k is the private key. While all parties need to keep k private as before, pk can be made public. Looking ahead, this will help us address the difficulty of computing ciphertexts in a distributed manner.

To be more concrete, it is sufficient to address the following issues.

- For each wire w, all parties need to jointly prepare two key pairs where the private keys are additively shared among all parties.
- We need to design a protocol that allows all parties to efficiently compute a ciphertext when pk is known to all parties but the message m is additively shared.

For simplicity, we start with the semi-honest security. We will discuss how to upgrade our protocol to achieve malicious security in Sect. 2.4.

Addressing the First Difficulty. For the first difficulty, we note that it is sufficient to use a public-key encryption scheme that is linearly homomorphic for public keys: For two key pairs (pk_1, k_1) and (pk_2, k_2), $(\mathsf{pk}_1 \tilde{+} \mathsf{pk}_2, k_1 + k_2)$ is also a valid key pair. Here $\tilde{+}$ refers to the homomorphic operation defined by the public-key encryption scheme.

Now to generate (pk, k) such that k is additively shared among all parties,

1. Each party P_i generates (pk_i, k_i) and sends pk_i to the first party P_1;

2. P_1 locally computes $\mathsf{pk} = \mathsf{pk}_1 \tilde{+} \mathsf{pk}_2 \tilde{+} \ldots \tilde{+} \mathsf{pk}_n$ and sends pk to all parties.
3. Each party P_i views his private key k_i as an additive share of $k = k_1 + \ldots + k_n$ and outputs (pk, k_i).

Note that when there are $t = n - 1$ corrupted parties, the adversary will learn pk_i generated by the honest party P_i. However, this is fine since the adversary learning pk and public keys $\{\mathsf{pk}_j\}_{j \neq i}$ generated by all corrupted parties can anyway compute pk_i locally.

Addressing the Second Difficulty. For the second difficulty, although pk is public, the message m to be encrypted is still secret shared among all parties. It is unclear how to emulate the encryption algorithm in a black-box way.

Our main observation is that, to build a constant-round MPC protocol with linear communication, it is not necessary to obtain a single and compact ciphertext for the message m. Recall that in the evaluation phase, the evaluator will obtain the proper private key (wire label) for each wire and need to decrypt the corresponding ciphertext. We note that it is sufficient to let each party provide a separate ciphertext for his share of the message.

To be more concrete, suppose m is additively shared among all parties where each party P_i holds m_i. We let each party P_i encrypt his message share m_i using the public key pk and send the ciphertext ct_i to the evaluator. Now the evaluator with the private key k and $(\mathsf{ct}_1, \ldots, \mathsf{ct}_n)$ can already decrypt m_i from each ct_i and compute $m = m_1 + \ldots + m_n$. In this way, we can maintain linear communication while using the underlying public-key encryption scheme in a black-box way.

Summary of Our Solution. In summary, we will replace the symmetric-key encryption scheme in the BMR template by a public-key encryption scheme that is linearly homomorphic for public keys. When preparing the wire label, we let all parties prepare (pk, k) where pk is public and k is secretly shared among all parties. Then we follow the BMR template and compute the message m (which is the proper wire label (private key) for the next layer) to be encrypted, which is also secretly shared among all parties. We let each party locally encrypt his message share using pk and send the ciphertext to P_1. In the evaluation phase, P_1 will serve as the evaluator to compute and distribute the final output.

2.3 Concrete Instantiation of PKE

In our work, we instantiate the public-key encryption scheme by a variant of the well-known El-Gamal encryption scheme [18] based on the Decisional Diffie-Hellman (DDH) assumption. For a DDH group G with group generator g and order p, the key generation algorithm outputs a random value $k \in \{0, \ldots, p-1\}$ as the private key and $\mathsf{pk} = g^k$ as the public key. To encrypt a group element m, the encryption algorithm samples a random value $r \in \{0, \ldots, p-1\}$ and outputs $(g^r, \mathsf{pk}^r \cdot m)$. The security follows from the DDH assumption which states that when k and r are uniformly random, given $\mathsf{pk} = g^k$ and g^r, $\mathsf{pk}^r = g^{k \cdot r}$

is computationally indistinguishable from a random group element. Thus, pk^r serves as a random mask for m.

Firstly, note that the El-Gamal encryption scheme is already linearly homomorphic for public keys: for any two key pairs (pk_1, k_1) and (pk_2, k_2), $(\mathsf{pk}_1 \cdot \mathsf{pk}_2, k_1 + k_2)$ is also a valid key pair. However, the problem with using the El-Gamal encryption scheme is that it can only be used to encrypt a group element. On the other hand, for each gate in Yao's garbled circuit, the message m that we want to encrypt is a wire label of the output wire. Recall that the wire label corresponds to the private key of the public-key encryption scheme, which is within $\{0, \ldots, p-1\}$. Taking a closer look at this issue, although pk^r is indistinguishable from a random group element, we cannot view pk^r as a uniform string due to the algebraic structure of the DDH group.

We resolve this issue by converting pk^r to a random string relying on pseudorandom generator (PRG) so that it can be used as a one-time pad key to encrypt the message m (which is also viewed as a bit-string).

- Let Ext be a strong-seeded randomness extractor Ext and Prg be a pseudorandom generator Prg. We modify the encryption algorithm as follows: After computing pk^r, we apply Ext on pk^r to obtain a (pseudo)random output. Then we apply Prg to stretch the length of the random output and use the result to encrypt m. Thus, the ciphertext is defined by $(g^r, \mathsf{seed}, m \oplus \mathsf{Prg}(\mathsf{Ext}(\mathsf{pk}^r; \mathsf{seed})))$. In practice, one can replace $\mathsf{Prg}(\mathsf{Ext}(\cdot))$ by a random oracle for practical efficiency.

We note that, in the actual construction of Yao's garbled circuit, for each gate, each message needs to be encrypted under two public keys (one from each input wire). While a direct solution is to do the encryption using the two public keys one by one, in Sect. 4, we provide an optimized version that directly works for the two-public-key setting.

2.4 Towards Malicious Security

So far, we have mainly focused on semi-honest security. To achieve malicious security, we rely on the standard technique of message authentication codes (MAC) [16,17]. At the beginning of the protocol, all parties together hold an additive sharing of a global MAC key Δ, denoted by $[\Delta]$. A SPDZ sharing of a secret x is defined by a tuple of three additive sharings: $[\![x]\!] = ([x], [\Delta], [\Delta \cdot x])$. When the secret x is reconstructed, all parties can use a MAC check protocol [16] to verify the correctness of the reconstruction. We rely on the malicious variant of Le Mans [36] to support addition and multiplication operations over SPDZ sharings with linear communication complexity.

However, we also need to protect against the following malicious behaviors:

- Recall that to prepare a key pair (pk, k), all parties first prepare a SPDZ sharing $[\![k]\!]$. Then each party sends g^{k_i} to P_1, which allows P_1 to compute g^k and send it to all parties.

However, parties may end up with an incorrect public key pk' either due to a corrupted party P_i sending an incorrect g^{k_i} to P_1 or because P_1 is corrupted. In the worst case, the adversary may even learn the private key (the discrete log) of pk', and the security of the public-key encryption scheme is gone.
- When doing the encryption, a corrupted party may not encrypt his correct share of the message. Then in the evaluation phase, the evaluator may decrypt an incorrect message and obtain incorrect function outputs.
- When the evaluator is corrupted, he may not send the correct function outputs to all parties at the end of the protocol.

Handling the First Attack. For the first attack, all parties will together verify the correctness of public keys. Since $(\mathsf{pk}, [\![k]\!])$ is linearly homomorphic, we simply check a random linear combination of all key pairs.

More precisely, suppose all parties have prepared $\{(\mathsf{pk}^{(i)}, [\![k^{(i)}]\!])\}_{i=0}^{N}$. To verify the correctness of each $\mathsf{pk}^{(i)}$, all parties compute a random linear combination

$$(\mathsf{pk}, [\![k]\!]) = (\mathsf{pk}^{(0)}, [\![k^{(0)}]\!]) + \sum_{i=1}^{N} r_i \cdot (\mathsf{pk}^{(i)}, [\![k^{(i)}]\!]),$$

where $r_1, \ldots, r_N$ are (pseudo)random coefficients. Now it is sufficient to check whether pk is correct with respect to k. Relying on the MAC, all parties can verifiably reconstruct the secret k and then check whether $\mathsf{pk} = g^k$.

Handling the Last Two Attacks. We note that given a key pair (pk, k), one can verify whether it is valid by checking $\mathsf{pk} = g^k$. Since the public keys are known to all parties, an honest party can use this property to verify the correctness of a wire label (private key).

The second attack only occurs when the evaluator is honest. In the evaluation phase, whenever an honest evaluator computes a wire label, he can use the above approach to check the validity of the wire label. In case he does not obtain the correct wire label, the protocol will abort. In this way, the protocol will only proceed if an honest evaluator obtains the correct wire label for each wire.

For the third attack, we require the evaluator to send the wire label for the output wires to all parties so that an honest receiver can check the validity of the wire label locally. Note that a malicious evaluator can only learn the wire label corresponding to the function output. In this way, an honest receiver will not accept an incorrect function output.

Optimizations. We note that in the evaluation phase, for each wire, the evaluator will only learn one of the two wire labels associated with this wire. Thus, we may set the difference between these two wire labels to be the same for all wires. This trick has been used in many previous works for constructing efficient Yao's garbled circuits.

More concretely, for each wire w, recall that all parties need to prepare $(\mathsf{pk}_{w,0}, [k_{w,0}])$ and $(\mathsf{pk}_{w,1}, [k_{w,1}])$. We require that $k_{w,1} - k_{w,0} = \Delta$, where Δ is the MAC key of the underlying SPDZ protocol. This brings us two benefits.

- First, when computing $\mathsf{pk}_{w,0}$ and $\mathsf{pk}_{w,1}$, it is sufficient to compute g^{Δ} and $\mathsf{pk}_{w,0}$. Then all parties can locally compute $\mathsf{pk}_{w,1} = g^{\Delta} \cdot \mathsf{pk}_{w,0}$. In this way, we only need to prepare one key pair for each wire.
- Second, in the Yao's garbled circuit, recall that for each gate, we need to compute 4 ciphertexts. Let a, b denote the input wires of this gate and c denote the output wire. For all $i, j \in \{0, 1\}$, we need to first compute $\chi_{i,j}$, which is the index of the private key we need to encrypt under the public keys $\mathsf{pk}_{a,i}$ and $\mathsf{pk}_{b,j}$; i.e., the ciphertext we need to compute is $\mathsf{Enc}_{\mathsf{pk}_{a,i},\mathsf{pk}_{b,j}}(\chi_{i,j}, k_{c,\chi_{i,j}})$[1]. Then we need to compute an additive sharing of $k_{c,\chi_{i,j}}$. Usually, this is done by first computing a SPDZ sharing $[\![\chi_{i,j}]\!]$ and then using $\chi_{i,j}$ to choose one of the two private keys, which requires one additional multiplication operation. We observe that $k_{c,\chi_{i,j}} = k_{c,0} + \chi_{i,j} \cdot \Delta$. Note that all parties have already held $[\chi_{i,j} \cdot \Delta]$ from $[\![\chi_{i,j}]\!]$. Thus, all parties can locally compute $[k_{c,\chi_{i,j}}] = [k_{c,0}] + [\chi_{i,j} \cdot \Delta]$.

Remark 1 (Free XOR and the Assumption of Random Oracle). In our construction, we have to use a large prime field due to the DDH assumption. This unfortunately is not compatible with the free-XOR technique introduced in [33], which requires working over an extension field of the binary field. We leave the question of incorporating free-XOR into our technique to future work.

As in all previous works that use the same difference between the two labels for all wires, this optimization only works under the assumption of a random oracle due to the issue of circular encryption. For concrete efficiency, we will mainly focus on the construction with the above optimization assuming a random oracle and refer the readers to the full version of this paper for the one that does not require a random oracle.

2.5 Related Works

As we have mentioned above, the main efficiency bottleneck of the BMR construction is that it requires all parties to emulate the underlying encryption algorithm, which needs to use the underlying symmetric-key encryption scheme in a non-black-box way. For typical instantiations, the symmetric-key encryption scheme involves computation of a pseudo-random function (PRF). To be more concrete, to encrypt a message m with secret key k for a gate with identifier g, the ciphertext is defined by

$$\mathsf{ct} = \mathsf{PRF}_k(g) \oplus m.$$

Starting from secret sharings of k and m, the joint computation of the PRF would incur a large overhead in both communication and computation depending upon the circuit size of the PRF.

To overcome this issue, Damgård and Ishai [15] proposed a variant of the BMR construction that uses PRG in a black-box way. At a high level, instead

[1] In our actual construction, we only encrypt $k_{c,\chi_{i,j}}$ but not $\chi_{i,j}$ since the evaluator can learn $\chi_{i,j}$ by comparing $k_{c,\chi_{i,j}}$ with the two public keys $\mathsf{pk}_{c,0}, \mathsf{pk}_{c,1}$.

of viewing that all parties hold a secret sharing for each wire label, we simply take the concatenation of all shares as the wire label $k = k_1 \| k_2 \| \ldots \| k_n$. Now the ciphertext is defined by

$$\mathsf{ct} = \mathsf{PRF}_{k_1}(g) \oplus \mathsf{PRF}_{k_2}(g) \oplus \ldots \oplus \mathsf{PRF}_{k_n}(g) \oplus m.$$

In this way, each party can locally apply PRF on his share and only secret share the result. However, since the size of each wire label is increased by a factor of n because of concatenation, both the size of the garbled circuit and the overall communication complexity are increased by a factor of n, i.e., $O(|C|n)$ for the size of the garbled circuit with $O(|C|n^2)$ communication!

A line of works focuses on partially resolving this issue by either considering a weaker security where there are more than 1 honest parties [3,23,24] or only reducing the size of the garbled circuit relying on advanced cryptographic tools and assumptions [4,6]. We note that [6] is also based on the DDH assumption and [4] has the potential of achieving overall linear communication complexity, which we will elaborate on below.

Comparison with [6]. The protocol in [6] follows the BMR framework to encrypt secret-shared messages with secret-shared keys. By using key-homomorphic PRF, the size of the garbled circuit is independent of the number of parties. However when using DDH, the key-homomorphic PRF is multiplicative homomorphic which requires them to multiply n values, one held by each party. This step incurs a quadratic communication. Besides, [6] only works against semi-honest adversaries.

Our work uses PKE and sacrifices the succinctness of the garbled circuit (i.e., the size is linear in the number of parties) to achieve linear communication.

Achieving Linear Communication From [4]. To reduce the size of the garbled circuit, the work [4] relies on a symmetric-key encryption scheme based on the LPN assumption. At a high level, to encrypt a message m with secret key k, the encryption algorithm is defined by

$$\mathsf{ct} = (\boldsymbol{A} \cdot k) \oplus e \oplus (\boldsymbol{M} \cdot m),$$

where $\boldsymbol{A}$ and $\boldsymbol{M}$ are public matrices, k, m are viewed as vectors of bits, and e is a bit-string (or a vector of bits) with each bit independently drawn from some Bernoulli distribution. Intuitively, the LPN assumption states that for a random string k, given $\boldsymbol{A}$, $(\boldsymbol{A} \cdot k) \oplus e$ is computationally indistinguishable from a uniform string, which is used as the one-time pad key to encrypt the message. To decrypt ct with k, one can compute $\mathsf{ct} \oplus (\boldsymbol{A} \cdot k)$ which is equal to $e \oplus (\boldsymbol{M} \cdot m)$. The matrix $\boldsymbol{M}$ is used to encode the message m so that the message can be recovered even if some bits of the codeword are incorrect due to the error string e. Note that the key size and the ciphertext size do not grow with the number of parties, thus achieving $O(|C|)$ for the size of the garbled circuit.

The main benefit of this encryption scheme is that, if parties have additive sharings of k and m, and can generate additive sharings of e, then the encryption algorithm can be computed via local computation. In [4], the generation of additive sharings of error strings and the computation of secret sharings of k and m are done via the SPDZ protocol in a black-box way. When instantiating the underlying SPDZ protocol by Le Mans [36], it is possible to achieve linear communication $O(|C|n)$[2].

Despite the potential of achieving linear communication, the LPN assumption usually requires the use of a very large parameter to achieve the desired level of security. For example, the analysis in [4] shows that when the computational security parameter $\lambda = 128$ and the statistical security parameter $\kappa = 80$, the ciphertext size needs to be $\ell = 8925$ bits. Furthermore, all parties need to jointly prepare ℓ random bits that follow the Bernoulli distribution to obtain e for each encryption. In [4], to prepare one random bit sharing such that the secret is 1 with probability τ, all parties first generate $\log(1/\tau)$ random bit sharings where the secrets are uniformly distributed and then multiply them together. As a result, there is a large constant overhead in [4].

Implicit Overhead of LPN-Based Encryption. We note the symmetric-key encryption scheme used in [4] satisfies an even stronger homomorphism: Each party can locally encrypt his message share using his key share. Then the summation of all ciphertexts corresponds to a valid ciphertext of m under the secret key k. This seems to give a way to avoid generating the error string e jointly. Instead, each party P_i can generate his own error string e_i (used to encrypt his message share) and the final error string becomes $e = \bigoplus_{i=1}^{n} e_i$.

However, even if all parties perform honestly, note that the error accumulates in the LPN-based encryption scheme and this requires the ciphertext to be long enough so that one can still recover the message during the decryption phase. This would implicitly require the ciphertext size to be linear in n. This is why in both [3,4], the error string is generated jointly.

A similar issue would occur if one tries to instantiate the public-key encryption scheme used in our construction from the LPN assumption: The error accumulated during the aggregation of all parties' public keys may require the key size to be proportional to the number of parties, resulting in a quadratic communication overhead.

Comparison with [20]*.* We note that a recent work [20] also achieves linear communication for computing the garbled circuit and the garbled circuit size is independent of the number of parties. However, their construction only focuses on semi-honest security and requires a threshold homomorphic encryption scheme, while our protocol achieves malicious security with lighter assumptions. In addition, their construction requires $O(\log n)$ rounds to compute the garbled circuits.

[2] We note that [4] actually needs to compute binary circuits using the SPDZ protocol. However, the malicious variant of Le Mans [36] currently only works for a large finite field. In this work, we omit this distinction when comparing [4] with our result.

3 Preliminaries

Notations. Let $\mathbb{F}_p$ be a prime field of order p. Let U_m be the uniform distribution over $\{0,1\}^m$. We use κ to denote the statistical security parameter, and we use λ to denote the computational security parameter.

3.1 Basic Definitions and Primitives

Definition 1 (Statistical Distance). *For random variables X and Y taking values in $\mathcal{X}$, their statistical difference is defined as*

$$\varDelta(X,Y) = \max_{T \subset \mathcal{X}} |\Pr[X \in T] - \Pr[Y \in T]|.$$

We say that X and Y are ϵ-close if $\varDelta(X,Y) \leq \epsilon$.

An extractor [32] can be used to extract uniform randomness out of a weakly random value which is only assumed to have sufficient min-entropy (see the full version for more about min-entropy and k-sources). A strong seeded extractor is defined as follows.

Definition 2 (Strong Seeded Extractors [32]). *An efficient function $\mathsf{Ext} : \mathcal{X} \times \{0,1\}^d \rightarrow \{0,1\}^m$ is a strong (k,ϵ)-extractor if for every k-source X (i.e., if $\forall x, \Pr[X = x] \leq 2^{-k}$) on $\mathcal{X}$, $(U_d, \mathsf{Ext}(X, U_d))$ is ϵ-close to (U_d, U_m).*

A strong seeded extractor can be constructed with an efficient universal hash function [12,31,33,37] following the Leftover Hash Lemma [22], as stated below.

Theorem 2. *There exists a strong $(k, 2^{-\kappa})$-seeded extractor $\mathsf{Ext} : \mathcal{X} \times \{0,1\}^k \rightarrow \{0,1\}^{k-2\kappa}$.*

Definition 3. (Computational Distance). *For random variables X and Y taking values in $\mathcal{X}$, the advantage of a circuit D in distinguishing X and Y is defined as*

$$\varDelta_D(X,Y) = |\Pr[D(X) = 1] - \Pr[D(Y) = 1]|.$$

Let $\mathcal{D}_t$ be the set of all probabilistic circuits of size t. The computational difference of X and Y is defined as $\mathsf{CD}_t(X,Y) = \max_{D \in \mathcal{D}_t} \varDelta_D(X,Y)$. When $X = X_\lambda$ and $Y = Y_\lambda$ are families of distributions indexed by a security parameter λ, we say that X and Y are computationally indistinguishable, denoted $X =_c Y$, if for every polynomial $t(\cdot)$, $\mathsf{CD}_{t(\lambda)}(X,Y) = \mathsf{negl}(\lambda)$.

Definition 4. (Pseudorandom Generator). *A length-increasing function $\mathsf{Prg} : \{0,1\}^\lambda \rightarrow \{0,1\}^m$ is a pseudorandom generator (PRG) if $\mathsf{Prg}(U_\lambda) =_c U_m$.*

Decisional Diffie-Hellman (DDH) Assumption. We assume that the DDH assumption holds, i.e., for a group G of a 2λ-bit prime order p, let g be a generator of G, then:

$$\{g^a, g^b, g^{ab} : a, b \xleftarrow{\$} \mathbb{F}_p\} =_c \{g^a, g^b, g^c : a, b, c \xleftarrow{\$} \mathbb{F}_p\}.$$

Security Model. We define the security of multiparty computation in the *real and ideal world paradigm* [11]. Informally, we consider a protocol Π to be secure if any adversary's view in its execution in the real world can also be simulated in the ideal world. For more details, we refer the readers to the full version.

3.2 Secret Sharing

Let $[x]$ denote an unauthenticated additive sharing of x with P_i's share $x^{(i)}$. We use $\mathcal{P} = \{P_i\}_{i=1}^n$ to denote the set of all parties. For all $\mathcal{P}_A, \mathcal{P}_B \subset \mathcal{P}$, we follow the definition in [36] to define an authenticated sharing $\langle x \rangle^{\mathcal{P}_A, \mathcal{P}_B}$ as follows.

- All parties in $\mathcal{P}_A$ together hold an additive sharing of x. Each party $P_j \in \mathcal{P}_B$ holds a global key $\Delta^{(j)}$.
- For every $P_i \in \mathcal{P}_A, P_j \in \mathcal{P}_B$, P_j holds a random local key $K_i^{(j)}$ and P_i holds the MAC of $x^{(i)}$ defined by $M_j^{(i)} = K_i^{(j)} + \Delta^{(j)} \cdot x^{(i)}$.

An authenticated additive sharing

$$\langle x \rangle^{\mathcal{P}_A, \mathcal{P}_B} = ((x^{(i)}, (M_j^{(i)})_{P_j \in \mathcal{P}_B})_{P_i \in \mathcal{P}_A}, (\Delta^{(j)}, (K_i^{(j)})_{P_i \in \mathcal{P}_A})_{P_j \in \mathcal{P}_B})$$

can be locally converted to a SPDZ sharing $[\![x]\!] = ([x], [\Delta], [\Delta \cdot x])$ when $\mathcal{P}_B = \mathcal{P}$ by letting each P_i compute

$$\big(x^{(i)}, \Delta^{(i)}, \Delta^{(i)} \cdot x^{(i)} + \sum_{j \neq i} (M_j^{(i)} - K_j^{(i)})\big),$$

where $x^{(i)} = M_j^{(i)} = K_i^{(j)} = 0$ for all $P_i \notin \mathcal{P}_A$ and $P_j \in \mathcal{P}$. When $\mathcal{P}_A = \mathcal{P}_B = \mathcal{P}$, we simply use $\langle x \rangle$ to denote $\langle x \rangle^{\mathcal{P}, \mathcal{P}}$.

3.3 Functionalities for Sub-Protocols

We borrow the following functionalities from [36].

Programmable OLE. We use a functionality for *random, programmable oblivious linear evaluation* (OLE), $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ (see Fig. 1). This is a two-party functionality, which computes a batch of secret-shared products, i.e. random tuples $(u_i, v_i), (x_i, w_i)$, where $w_i = u_i \cdot x_i + v_i$, over the field $\mathbb{F}_p$. The *programmability* requirement is that, for any given instance of the functionality, the party who obtains u_i or x_i can program these to be derived from a chosen random seed. This allows the same u_i, v_i to be used in different instances of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. We model the programmability with an expansion function Expand, which is a PRG.

Multiparty VOLE. *Vector oblivious linear evaluation* (VOLE) can be seen as a batch of OLEs with the same x_i value in each tuple, i.e., a vector $\boldsymbol{w} = \boldsymbol{u} \cdot x + \boldsymbol{v}$, where $x \in \mathbb{F}_p$ is a scalar given to one party. In the functionality of multiparty VOLE, $\mathcal{F}_{\mathsf{nVOLE}}$ (see Fig. 2), every pair of parties (P_i, P_j) is given a random VOLE instance $\boldsymbol{w}_j^{(i)} = \boldsymbol{u}^{(i)} \cdot x^{(j)} + \boldsymbol{v}_i^{(j)}$. The functionality guarantees that the same $\boldsymbol{u}^{(i)}$

Functionality $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$

Let $\mathsf{Expand} : S \to \mathbb{F}_p^m$ be an expansion function with seed space S and output length m. On receiving $s_a \in S$ from P_A and $s_b \in S$ from P_B:

1. The trusted party computes $\boldsymbol{u} = \mathsf{Expand}(s_a), \boldsymbol{x} = \mathsf{Expand}(s_b)$ and samples $\boldsymbol{v} \in \mathbb{F}_p^m$.
2. The trusted party outputs $\boldsymbol{w} = \boldsymbol{u} * \boldsymbol{x} + \boldsymbol{v}$ to P_A and $\boldsymbol{v}$ to P_B.

Corrupted Parties: If P_B is corrupt, $\boldsymbol{v}$ may be chosen by $\mathcal{S}$. For a corrupt P_A, $\mathcal{S}$ can choose $\boldsymbol{w}$ (and then $\boldsymbol{v}$ is recomputed accordingly).

Fig. 1. Functionality for programmable OLE.

or $x^{(j)}$ values will be used in each instance involving P_i or P_j. Unlike OLE, the $\boldsymbol{u}^{(i)}, x^{(i)}$ values in $\mathcal{F}_{\mathsf{nVOLE}}$ are not programmable, and the outputs to P_i is required to be a short seed which can be expanded to $\boldsymbol{u}^{(i)}$ so that P_i can later use this as an input to $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$.

Functionality $\mathcal{F}_{\mathsf{nVOLE}}$

Let $\mathsf{Expand} : S \to \mathbb{F}_p^m$ be an expansion function with seed space S and output length m.

Init: On receiving Init from P_i for $i \in [1, n]$, the trusted party samples $\Delta^{(i)} \in \mathbb{F}_p$, sends it to P_i, and ignores all subsequent Init commands from P_i.

Extend: On receiving Extend from every $P_i \in \mathcal{P}$:

1. For each honest P_i, the trusted party randomly samples $\mathsf{seed}^{(i)} \in S$.
2. For each P_i, let $\boldsymbol{u}^{(i)} = \mathsf{Expand}(\mathsf{seed}^{(i)})$. The trusted party samples $(\boldsymbol{v}_i^{(j)})_{j \neq i} \in \mathbb{F}_p^m$, retrieves $\Delta^{(j)}$ and computes $\boldsymbol{w}_j^{(i)} = \boldsymbol{u}^{(i)} \cdot \Delta^{(j)} + \boldsymbol{v}_i^{(j)}$.
3. If P_j is corrupt, $\mathcal{S}$ can send a set I to the trusted party. If $\mathsf{seed}^{(i)} \in I$, the trusted party sends $\mathsf{success}$ to P_j and continues. Otherwise, the trusted party sends abort to both parties, outputs $\mathsf{seed}^{(i)}$ to P_j and aborts.
4. The trusted party outputs $((\mathsf{seed}^{(i)}, (\boldsymbol{w}_j^{(i)}, \boldsymbol{v}_j^{(i)})_{j \neq i})$ to P_i.

Corrupted Parties: A corrupt P_i can choose $\Delta^{(i)}$ and $\mathsf{seed}^{(i)}$. It can also choose $\boldsymbol{w}_j^{(i)}$ (and then $\boldsymbol{v}_i^{(j)}$ is recomputed accordingly) and $\boldsymbol{v}_j^{(i)}$.

Global key query: If P_i is corrupted, the trusted party receives $(\mathsf{guess}, \boldsymbol{\Delta}')$ from $\mathcal{S}$ with $\boldsymbol{\Delta}' \in \mathbb{F}_p^n$. If $\boldsymbol{\Delta}' = \boldsymbol{\Delta}$ where $\boldsymbol{\Delta}' = (\Delta^{(1)}, \dots, \Delta^{(n)})$, the trusted party sends $\mathsf{success}$ to P_i and ignores any subsequent global key query, otherwise, the trusted party sends $(\mathsf{abort}, \boldsymbol{\Delta})$ to P_i, abort to P_j and aborts.

Fig. 2. Functionality for n-party VOLE.

We also use the standard functionalities $\mathcal{F}_{\mathsf{Coin}}$ (to generate a random common coin in $\mathbb{F}_p$) and $\mathcal{F}_{\mathsf{Commit}}$ (to model commitment a scheme), we refer the readers to the full version for more details.

3.4 MAC Check on Opened Values

For a SPDZ sharing $[\![x]\!]$, when we say the parties open $[\![x]\!]$, we mean that the parties run the following Π_{Open} protocol on $[\![x]\!]$.

Protocol $\Pi_{\mathsf{Open}}([\![x]\!])$

1. All the parties send their shares of $[x]$ to P_1.
2. P_1 reconstructs x and sends it to all the parties.

Fig. 3. Protocol to open a SPDZ sharing.

When the secrets of some SPDZ sharings are opened to all parties, they can check the correctness relying on the MACs by a standard SPDZ MAC check protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ [16] in the $\{\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model (see Fig. 4).

Protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$

Let all parties agree on a PRG $\mathsf{Prg} : \mathbb{F}_p \to \mathbb{F}_p^m$. Parties want to check the MACs on opened values $(A_1, \ldots, A_m)$ for sharings $([\![A_1]\!], \ldots, [\![A_m]\!])$.

1. The parties call $\mathcal{F}_{\mathsf{Coin}}$ to get a random seed in $\mathbb{F}_p$ and expand it with Prg to get random random values $\chi_1, \ldots, \chi_m \in \mathbb{F}_p$.
2. The parties compute $A = \sum_{i=1}^m \chi_i \cdot A_i$ and $[\gamma] = \sum_{i=1}^m \chi_i \cdot [\Delta \cdot A_i]$.
3. The parties compute $[\sigma] = [\gamma] - [\Delta] \cdot A$. Each P_i calls $\mathcal{F}_{\mathsf{Commit}}$ with input $(\mathsf{commit}, P_i, [\sigma], \tau_{[\sigma]})$.
4. The parties open their commitments and check that $\sum_{i=1}^n [\sigma] = 0$. If not, the parties output abort and abort the protocol.

Fig. 4. Protocol for MAC checking.

4 Encryption Scheme Based on DDH

In this section, we construct a public-key encryption scheme based on the DDH assumption. We will first give a construction based on strong seeded extractors, and then simplify it under the assumption of random oracles.

4.1 Encryption Scheme Based on Strong Seeded Extractors

We now introduce our construction of an encryption scheme based on strong seeded extractors. Let p be a prime number and $\mathbb{F}_p$ be the prime field of size p. Our goal is to encrypt a message in $\mathbb{F}_p$.

Let $\oplus$ denote the bit-wise XOR of two binary strings of the same length. Let $k = \max\{2\lambda, \lambda + 2\kappa\}$. Our encryption scheme PKE_1 is a tuple of four PPT algorithms $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ defined as follows:

- $\mathsf{Setup}(\lambda, \kappa)$: The setup algorithm Setup samples the following:
  1. A group G of order p with a generator g, where p is a k-bit prime. Let ℓ denote the length of group elements in G.
  2. A strong $(k, 2^{-\kappa})$-extractor $\mathsf{Ext} : \{0,1\}^{\ell} \times \{0,1\}^{k} \to \{0,1\}^{\lambda}$ (Theorem 2).
  3. A pseudorandom generator $\mathsf{Prg} : \{0,1\}^{\lambda} \to \{0,1\}^{k}$.

  Setup outputs public parameters $pp = (G, p, g, \mathsf{Ext}, \mathsf{Prg}, \lambda, \kappa)$.
- $\mathsf{Gen}(pp)$: The key-generation algorithm Gen samples $\mathsf{sk}_1, \mathsf{sk}_2 \in \mathbb{F}_p$ and computes $\mathsf{pk}_1 = g^{\mathsf{sk}_1}$, $\mathsf{pk}_2 = g^{\mathsf{sk}_2}$. Gen outputs $(\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2)$, where $\mathsf{sk}_1, \mathsf{sk}_2$ are the secret keys, and $\mathsf{pk}_1, \mathsf{pk}_2$ are the public keys.
- $\mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m)$: The encryption algorithm Enc runs as follows:
  1. Sample random $k_1, k_2 \in \mathbb{F}_p$, and compute $g^{k_1}, g^{k_2}, (\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2}$.
  2. Sample random $s \in \{0,1\}^{k}$, compute $m' = \mathsf{Prg}(\mathsf{Ext}((\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2}, s))$.
  3. Encode m to a vector in $\{0,1\}^{k}$. Output $c = (m \oplus m', s, g^{k_1}, g^{k_2})$.
- $\mathsf{Dec}(pp, \mathsf{sk}_1, \mathsf{sk}_2, c)$: Suppose $c = (m^*, s, g_1, g_2)$. Then, the decryption algorithm Dec outputs $m = m^* \oplus \mathsf{Prg}(\mathsf{Ext}(g_1^{\mathsf{sk}_1} \cdot g_2^{\mathsf{sk}_2}, s))$.

This public key encryption scheme guarantees that a party can decrypt a ciphertext with both secret keys, but he does not learn any information about the message with only one key. We state this as the following theorem, and we provide a proof of this theorem in the full version of this paper.

Theorem 3. $\mathsf{PKE}_1 = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *satisfies the following conditions:*

- **Correctness.** *Let* $pp \leftarrow \mathsf{Setup}(\lambda, \kappa)$*. For any message* $m \in \mathbb{F}_p$*,*

$$\Pr\left[\mathsf{Dec}(pp, \mathsf{sk}_1, \mathsf{sk}_2, c) = m : \begin{array}{l} (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp), \\ c \leftarrow \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m) \end{array}\right] = 1.$$

- **Security.** *Let* $pp \leftarrow \mathsf{Setup}(\lambda, \kappa)$*. Assume the DDH assumption over* G *with group generator* g*. Then for any pair of messages* $m_0, m_1 \in \mathbb{F}_p$*,*

$$\begin{aligned} &\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ =_c\ &\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ &\qquad\qquad\qquad\textit{and} \\ &\{\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ =_c\ &\{\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}. \end{aligned}$$

Instantiation of the Group. In practice, we can choose G to be an elliptic curve $E(\mathbb{F}_q)$ of size p. Each point on this elliptic curve can be expressed as a point in $\mathbb{F}_q^2$. Thus, we can take $\ell = 2 \log q$, where it is ensured by the Mordell-Weil Theorem (see [30,40]) that $\left| |E(\mathbb{F}_q)| - q - 1 \right| < 2\sqrt{q}$, which means $q = O(p)$.

4.2 Encryption Scheme Based on Random Oracle

If we assume the existence of a random oracle, the encryption scheme can be much simpler. We provide an encryption scheme $\mathsf{PKE}_2 = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$:

- $\mathsf{Setup}(\lambda)$: The setup algorithm Setup samples a group G of order p with a generator g, where p is a 2λ-bit prime. Then, Setup initializes a random oracle $\mathcal{O}$ with output length 2λ. Setup outputs $pp = (G, p, g, \mathcal{O}, \lambda)$.
- $\mathsf{Gen}(pp)$: The key-generation algorithm Gen samples $\mathsf{sk}_1, \mathsf{sk}_2 \in \mathbb{F}_p$ and computes $\mathsf{pk}_1 = g^{\mathsf{sk}_1}$, $\mathsf{pk}_2 = g^{\mathsf{sk}_2}$. Gen outputs $(\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2)$, where $\mathsf{sk}_1, \mathsf{sk}_2$ are secret keys, and $\mathsf{pk}_1, \mathsf{pk}_2$ are public keys.
- $\mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m)$: The encryption algorithm Enc runs as follows:
  1. Sample random $k_1, k_2 \in \mathbb{F}_p$, then compute g^{k_1}, g^{k_2}, and $(\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2}$.
  2. Query the random oracle $\mathcal{O}$ with input $(\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2}$ to obtain $m' = \mathcal{O}((\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2})$.
  3. Encode m to a vector in $\{0,1\}^{2\lambda}$. Output $c = (m \oplus m', g^{k_1}, g^{k_2})$.
- $\mathsf{Dec}(pp, \mathsf{sk}_1, \mathsf{sk}_2, c)$: Suppose $c = (m^*, g_1, g_2)$. Then, the decryption algorithm Dec outputs $m = m^* \oplus \mathcal{O}(g_1^{\mathsf{sk}_1} \cdot g_2^{\mathsf{sk}_2})$.

Similarly to PKE_1, PKE_2 satisfies the correctness and security properties. A proof of the following theorem is provided in the full version of this paper.

Theorem 4. *Assume the DDH assumption over G with group generator g and random oracles,* $\mathsf{PKE}_2 = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *satisfies the same* **Correctness** *and* **Security** *conditions as* PKE_1.

5 Preprocessing Phase

5.1 Preprocessing Functionality

In the preprocessing phase, all the parties need to prepare several sharings to be used in the main protocol. The preprocessing functionality $\mathcal{F}_{\mathsf{prep}}$ is defined in Fig. 5. It allows the parties to prepare the following sharings:

- **Random Values:** A SPDZ sharing $[\![r]\!]$ of a random element $r \in \mathbb{F}_p$.
- **Triples:** SPDZ sharings $[\![a]\!], [\![b]\!], [\![c]\!]$ for $c = a \cdot b$, where a, b are random elements in $\mathbb{F}_p$.
- **Random Bits:** SPDZ sharings $[\![\lambda]\!]$, where $\lambda \in \{0,1\} \subset \mathbb{F}_p$ is a random bit.

With the authenticated triples, the parties can do multiplication following the SPDZ protocol. We present Π_{Mult} in Fig. 6.

5.2 Preprocessing Protocol

For simplicity, we use $\boldsymbol{x}[k]$ to denote the k-th entry of $\boldsymbol{x}$. In Fig. 7, we provide our protocol Π_{prep} realizing the preprocessing functionality $\mathcal{F}_{\mathsf{prep}}$ in the $\{\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{nVOLE}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model.

Functionality $\mathcal{F}_{\mathsf{prep}}$

Init: On receiving $(\mathsf{Init}, m_T, m_R, m_B)$ from every P_i, the trusted party samples a random element in $\mathbb{F}_p$ as Δ and sends g^{Δ} to $\mathcal{S}$. Then:

1. For each corrupted party P_i, the trusted party receives $\Delta^{(i)}$ from $\mathcal{S}$ and sends it to P_i.
2. The trusted party samples a MAC key $\Delta^{(i)} \leftarrow \mathbb{F}_p$ for each honest party P_i such that $\sum_{i=1}^{n} \Delta^{(i)} = \Delta$. Then, it sends $\Delta^{(i)}$ to each honest party P_i.

Finally, the trusted party sends g^{Δ} to all the parties and ignores subsequent Init commands from each P_i.

Random Values: Repeat the following m_R times:

1. The trusted party randomly samples $r \in \mathbb{F}_p$ and computes $\Delta \cdot r$.
2. The trusted party receives corrupted parties' shares of $[\![r]\!]$ from $\mathcal{S}$ and randomly samples honest parties' shares of $[\![r]\!]$ based on the secret and corrupted parties' shares.
3. The trusted party outputs $[\![r]\!]$ to the parties.

Triples: Repeat the following m_T times:

1. Run steps 1 and 2 from **Random Values** twice, to create sharings $[\![a]\!], [\![b]\!]$.
2. Let $c = a \cdot b$. The trusted party computes $\Delta \cdot c$.
3. The trusted party receives corrupted parties' shares of $[\![c]\!]$ from $\mathcal{S}$ and randomly samples honest parties' shares of $[\![c]\!]$ based on the secret and corrupted parties' shares.
4. The trusted party outputs $([\![a]\!], [\![b]\!], [\![c]\!])$ to the parties.

Random Bits: Repeat the following m_B times:

1. The trusted party randomly samples $\lambda \in \{0, 1\}$.
2. The trusted party receives corrupted parties' shares of $[\![\lambda]\!]$ from $\mathcal{S}$ and randomly samples honest parties' shares of $[\![\lambda]\!]$ based on the secret and corrupted parties' shares.
3. The trusted party outputs $[\![\lambda]\!]$ to the parties.

Abort. Upon receiving abort from $\mathcal{S}$, the trusted party sends Δ to $\mathcal{S}$, outputs abort to all the parties, and aborts.

Fig. 5. Functionality for preprocessing.

Each random SPDZ sharing is obtained by conversion from an authenticated additive sharing. The authenticated additive sharings are generated by $\mathcal{F}_{\mathsf{nVOLE}}$. More concretely, the parties invoke $\mathcal{F}_{\mathsf{nVOLE}}$ to receive the seeds and pair-wise MACs. Then, the parties expand their own seeds to get their shares of random additive sharings. We will generate m_R random sharings together with another random sharing for verification of g^{Δ}. The protocol Π_{rand} for generating random SPDZ sharings is given in Fig. 8.

To prepare multiplication triples, the parties prepare two seeds for random SPDZ sharings and then pair-wise invoke $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with their seeds, which enables each pair of parties to obtain a two-party additive sharing of the product of their shares of $[\boldsymbol{a}], [\boldsymbol{b}]$ expanded from their seeds. Adding all these shares together, the parties get unauthenticated additive sharings of $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. To convert an unauthenticated additive sharing $[c]$ to the SPDZ sharing $[\![c]\!]$, the parties need

Protocol $\Pi_{\mathsf{Mult}}([\![x]\!],[\![y]\!])$

To compute $[\![z]\!]=[\![x]\!]\cdot[\![y]\!]$:
1. Using a triple $([\![a]\!],[\![b]\!],[\![c]\!])$ generated using **Triples** from $\mathcal{F}_{\mathsf{prep}}$, the parties compute and open $[\![e]\!]=[\![x-a]\!],[\![d]\!]=[\![y-b]\!]$ by Π_{Open}.
2. The parties compute the multiplication by $[\![z]\!]=e\cdot d+e\cdot[\![b]\!]+d\cdot[\![a]\!]+[\![c]\!]$.

Fig. 6. Protocol for SPDZ multiplication.

Protocol Π_{prep}

Let $\mathsf{Expand}:S\to\mathbb{F}_p^m$ be an expansion function with seed space S and output length $m=\max\{m_T,m_R+1,m_B\}$. Each call of $\mathcal{F}_{\mathsf{nVOLE}}$ or $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ is with respect to this Expand and m.
Let G be a multiplicative group of order p with generator g.
Init:
1. Each $P_i\in\mathcal{P}$ sends Init to $\mathcal{F}_{\mathsf{nVOLE}}$ and receives $\Delta^{(i)}$.
2. Each P_i computes $g^{\Delta^{(i)}}$ and sends it to all the parties.
3. Each party locally computes $g^{\Delta}=\prod_{i=1}^{n}g^{\Delta^{(i)}}$.

The following protocols are then executed in order: $\Pi_{\mathsf{rand}},\Pi_{\mathsf{trip}},\Pi_{\mathsf{bit}},\Pi_{\mathsf{ver}}$.

Fig. 7. Protocol for preprocessing.

another random sharing $[\![\ell]\!]$, so that they can open $c+\ell$ and compute $[\![c]\!]=(c+\ell)-[\![\ell]\!]$. Since the corrupted parties may not send the correct seeds to $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, we need a verification. We utilize the technique from MASCOT [26] to use an extra triple $[\![a']\!],[\![b]\!],[\![c']\!]$ to verify the correctness of $c=a\cdot b$ for each triple $[\![a]\!],[\![b]\!],[\![c]\!]$. The protocol Π_{trip} for triple generation is given in Fig. 9.

To prepare random bit sharings, we apply the techniques of [14]. In particular, the parties prepare a pair of SPDZ sharings $[\![r]\!],[\![r^2]\!]$ in a similar way as preparing triples, where $r\in\mathbb{F}_p$ is random. The parties then open r^2 and compute $\sqrt{r^2}\in[0,\ldots,(p-1)/2]$. If $\sqrt{r^2}\neq 0$, $\sqrt{r^2}$ is either r or $-r$, with equal probability.

Protocol Π_{rand}

Random Values:
- *Setup:* Each party P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ with input Extend. P_i receives $(s_r^{(i)},(\boldsymbol{M}_j^{(i)},\boldsymbol{K}_j^{(i)})_{j\neq i})$. Each P_i's $\mathsf{Expand}(s_r^{(i)}),\Delta^{(i)},(\boldsymbol{M}_j^{(i)},\boldsymbol{K}_j^{(i)})_{j\neq i}$ form his shares of a vector of m sharings $\langle\boldsymbol{r}\rangle$.
- To get the k-th random sharing $(1\leq k\leq m_R+1)$, each party takes the k-th share from $\langle\boldsymbol{r}\rangle$ and locally convert it to $[\![r]\!]$.

Fig. 8. Protocol for preparing random sharings.

Protocol Π_{trip}

Triples:

- *Setup:*
  1. Each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ 5 times, with input Extend, and receives the seeds $s_a^{(i)}, s_b^{(i)}, s_a^{(i)'}, s_\ell^{(i)}, s_\ell^{(i)'}$. The outputs define vectors of shares $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle, \langle \boldsymbol{a}' \rangle, \langle \boldsymbol{\ell} \rangle, \langle \boldsymbol{\ell}' \rangle$ such that $\boldsymbol{a}^{(i)} = \mathsf{Expand}(s_a^{(i)})$, $\boldsymbol{b}^{(i)} = \mathsf{Expand}(s_b^{(i)})$, $\boldsymbol{a}^{(i)'} = \mathsf{Expand}(s_a^{(i)'})$, $\boldsymbol{\ell}^{(i)} = \mathsf{Expand}(s_\ell^{(i)})$ and $\boldsymbol{\ell}^{(i)'} = \mathsf{Expand}(s_\ell^{(i)'})$.
  2. Every pair of parties (P_i, P_j) calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending $s_a^{(i)}$ and P_j sending $s_b^{(i)}$, and it sends back $\boldsymbol{u}_{i,j}$ to P_i and $\boldsymbol{v}_{j,i}$ to P_j, such that $\boldsymbol{u}_{i,j} + \boldsymbol{v}_{j,i} = \boldsymbol{a}^{(i)} * \boldsymbol{b}^{(j)}$.
  3. Every pair of parties (P_i, P_j) calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending $s_a^{(i)'}$ and P_j sending $s_b^{(i)}$, and it sends back $\boldsymbol{u}'_{i,j}$ to P_i and $\boldsymbol{v}'_{j,i}$ to P_j, such that $\boldsymbol{u}'_{i,j} + \boldsymbol{v}'_{j,i} = \boldsymbol{a}^{(i)'} * \boldsymbol{b}^{(j)}$.
- To get the k-th triple ($1 \le k \le m_T$):
  1. Let $\langle a_k \rangle, \langle b_k \rangle, \langle \ell_k \rangle$ be the k-th shares from $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle$ and $\langle \boldsymbol{\ell} \rangle$. Each P_i computes $c_k^{(i)} = a_k^{(i)} \cdot b_k^{(i)} + \sum_{j \ne i} (\boldsymbol{u}_{i,j}[k] + \boldsymbol{v}_{i,j}[k])$ as his share of $[c]$. Then the parties locally convert $\langle a_k \rangle, \langle b_k \rangle, \langle \ell_k \rangle$ to $[\![a_k]\!], [\![b_k]\!], [\![\ell_k]\!]$.
  2. The parties compute their shares of $[\ell_k + c_k] = [\ell_k] + [c_k]$ and send them to P_1. P_1 reconstructs $\ell_k + c_k$ and sends it to all the parties.
  3. Each P_i locally computes $[\![c_k]\!] = (\ell_k + c_k) - [\![\ell_k]\!]$. Then the parties get the triple $([\![a_k]\!], [\![b_k]\!], [\![c_k]\!])$.
  4. The parties repeat steps 1-3 on $\langle \boldsymbol{a}' \rangle, \langle \boldsymbol{b} \rangle, \langle \boldsymbol{\ell}' \rangle$ in place of $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle, \langle \boldsymbol{\ell} \rangle$ to obtain $([\![a'_k]\!], [\![b_k]\!], [\![c'_k]\!])$.
  5. The parties output $([\![a_k]\!], [\![b_k]\!], [\![c_k]\!])$.

Fig. 9. Protocol for generating multiplication triples.

Then, $2^{-1}((\sqrt{r^2})^{-1} \cdot r + 1)$ is uniformly random in $\{0, 1\}$. The protocol Π_{bit} for generating random bit sharings is given in Fig. 10.

We need to verify the correctness of g^Δ, $c = a \cdot b$ for each triple, and the correctness of $[\![r^2]\!]$ for preparing bit sharings. For g^Δ, we sacrifice a random SPDZ sharing $[\![r]\!] = ([r], [\Delta], [\Delta \cdot r])$. Then, we can verify g^Δ by verifying that the product of all the parties' shares of $(g^\Delta)^{[r]} \cdot g^{-[\Delta \cdot r]}$ is equal to 1. To verify $c = a \cdot b$ for each triple, the parties sacrifice another triple $[\![a']\!], [\![b']\!], [\![c']\!]$ and use it to compute a SPDZ sharing $[\![\alpha \cdot ab - \alpha \cdot c]\!]$ for a random field element α, and this value is supposed to be opened as 0. Then we only need to check the opened values using $\Pi_{\mathsf{SPDZ\text{-}MAC}}$. The correctness of $[\![r^2]\!]$ can be verified in a similar way. The complete verification protocol Π_{ver} is given in Fig. 11.

Lemma 1. *The protocol Π_{prep} securely realizes $\mathcal{F}_{\mathsf{prep}}$ in the $\{\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{nVOLE}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model against a malicious adversary corrupting $n-1$ parties.*

We provide the proof of this lemma and a detailed analysis of communication in the full version. The communication cost of Π_{prep} is $(12nm_T + 16nm_B)\lambda$ bits.

Protocol Π_{bit}

Random Bits:

- *Setup:*
  1. Each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ 4 times, with input Extend, and receives the seeds $s_r^{(i)}, s_r^{(i)'}, s_\ell^{(i)}, s_\ell^{(i)'}$. The outputs define vectors of shares $\langle \boldsymbol{r}\rangle, \langle \boldsymbol{r'}\rangle, \langle \boldsymbol{\ell}\rangle, \langle \boldsymbol{\ell'}\rangle$ such that $\boldsymbol{r}^{(i)} = \mathsf{Expand}(s_r^{(i)})$, $\boldsymbol{r}^{(i)'} = \mathsf{Expand}(s_r^{(i)'})$, $\boldsymbol{\ell}^{(i)} = \mathsf{Expand}(s_\ell^{(i)})$ and $\boldsymbol{\ell}^{(i)'} = \mathsf{Expand}(s_\ell^{(i)'})$.
  2. Every unordered pair of parties (P_i, P_j) calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending $s_r^{(i)}$ and P_j sending $s_r^{(j)}$, and it sends back $\boldsymbol{u}_{i,j}$ to P_i and $\boldsymbol{u}_{j,i}$ to P_j, such that $\boldsymbol{u}_{i,j} + \boldsymbol{u}_{j,i} = \boldsymbol{r}^{(i)} * \boldsymbol{r}^{(j)}$.
  3. Every unordered pair of parties (P_i, P_j) calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending $s_r^{(i)'}$ and P_j sending $s_r^{(j)'}$, and it sends back $\boldsymbol{u}'_{i,j}$ to P_i and $\boldsymbol{u}'_{j,i}$ to P_j, such that $\boldsymbol{u}'_{i,j} + \boldsymbol{u}'_{j,i} = \boldsymbol{r}^{(i)'} * \boldsymbol{r}^{(j)'}$.
- To get the k-th sharing of a random bit ($1 \le k \le m_B$):
  1. Let $\langle r_k\rangle, \langle \ell_k\rangle$ be the k-th shares from $\langle \boldsymbol{r}\rangle, \langle \boldsymbol{\ell}\rangle$. Each P_i computes $R_k^{(i)} = (r_k^{(i)})^2 + 2\sum_{j\neq i} \boldsymbol{u}_{i,j}[k]$ as his share of $[R]$. Then the parties locally convert $\langle r_k\rangle, \langle \ell_k\rangle$ to $[\![r]\!], [\![\ell]\!]$.
  2. The parties send their shares of $[\ell_k + R_k]$ to P_1. P_1 reconstructs and sends $\ell_k + R_k$ to all the parties.
  3. Each P_i locally computes $[\![R_k]\!] = (\ell_k + R_k) - [\![\ell_k]\!]$. Then the parties get the pair $([\![r_k]\!], [\![R_k]\!])$.
  4. The parties repeat steps 1-3 on $\langle \boldsymbol{r}'\rangle, \langle \boldsymbol{\ell}'\rangle$ instead of $\langle \boldsymbol{r}\rangle$ and $\langle \boldsymbol{\ell}\rangle$ to obtain $([\![r'_k]\!], [\![R'_k]\!])$.
  5. The parties run Π_{Open} on $[\![R_k]\!]$. If $R_k = 0$, the parties output abort and abort the protocol. Otherwise, each party computes $\sqrt{R_k} \in [1, (p-1)/2]$.
  6. Each party computes their shares of $[\![\lambda_k]\!]$ by $[\![\lambda_k]\!] = 2^{-1} \cdot ((\sqrt{R_k})^{-1} \cdot [\![r_k]\!] + 1)$.
  7. The parties output $[\![\lambda_k]\!]$.

Fig. 10. Protocol for preparing random bit sharings.

6 Main Protocol

In this section, we provide our MPC protocol Π_{main} in the client-server model, where only clients have inputs and outputs. We assume that the clients are $C_1, \ldots, C_n$ and the servers are $S_1, \ldots, S_n$. The clients and servers run Π_{main} by running the following garbling phase and circuit evaluation phase in order.

Public Parameters. Let W be the number of wires, W_I be the number of input wires, and W_O be the number of output wires. Let G_A be the number of AND gates and G_X be the number of XOR gates. The public key encryption scheme we use is $\mathsf{PKE}_2 = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ from Sect. 4.2.

Theorem 5. *The protocol Π_{main} securely realizes $\mathcal{F}$ in the $\{\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model against a malicious adversary corrupting upto n clients and exactly $n-1$ servers.*

Protocol $\varPi_{\mathsf{ver}}$

Verification:

1. The parties sacrifice one SPDZ sharing $[\![r]\!]$ generated by **Random Values**. Let each P_i's share of $[r]$ be $r^{(i)}$, P_i's share of $[\varDelta \cdot r]$ be $m^{(i)}$.
2. Each party computes $\sigma^{(i)} = (g^{\varDelta})^{r^{(i)}} \cdot g^{-m^{(i)}}$ and calls $\mathcal{F}_{\mathsf{Commit}}$ with input $(\mathsf{commit}, P_i, \sigma^{(i)}, \tau_{\sigma^{(i)}})$.
3. The parties open their commitments and check that $\prod_{i=1}^{n} \sigma^{(i)} = 1$. If not, the parties output abort and abort the protocol.
4. The parties call $\mathcal{F}_{\mathsf{Coin}}$ to get a random value $\alpha \in \mathbb{F}_p$.
5. For each $([\![a_k]\!], [\![b_k]\!], [\![c_k]\!])$ and $([\![a'_k]\!], [\![b_k]\!], [\![c'_k]\!])$ prepared in **Triples**:
   (a) The parties compute and run $\varPi_{\mathsf{Open}}$ on $[\![e_k]\!] = [\![\alpha \cdot a_k - a'_k]\!]$.
   (b) The parties compute $[\![\tau_k]\!] = [\![\alpha \cdot c_k]\!] - e_k \cdot [\![b_k]\!] - [\![c'_k]\!]$
6. For each $([\![r_k]\!], [\![R_k]\!])$ and $([\![r'_k]\!], [\![R'_k]\!])$ prepared in **Random Bits**:
   (a) The parties compute and run $\varPi_{\mathsf{Open}}$ on $[\![d_k]\!] = [\![\alpha \cdot r_k - r'_k]\!]$.
   (b) The parties compute $[\![\tau'_k]\!] = [\![\alpha^2 \cdot R_k]\!] - d_k^2 - 2d_k \cdot [\![r'_k]\!] - [\![R'_k]\!]$.
7. The parties run $\varPi_{\mathsf{SPDZ\text{-}MAC}}$ to check the MACs on all the opened values, i.e., $\{e_k\}_{k \in [1,m_T]}, \{d_k, R_k\}_{k \in [1,m_B]}$.[1]
8. The parties run $\varPi_{\mathsf{SPDZ\text{-}MAC}}$ to check the MACs on $\{\tau_k\}_{k \in [1,m_T]}, \{\tau'_k\}_{k \in [1,m_B]}$ assuming that each $[\![\tau_k]\!]$ and $[\![\tau'_k]\!]$ has been opened to 0.

[1] Here we do not check the MAC on $\ell_k + c_k$ in the generation of triples and $\ell_k + R_k$ in the generation of random bits. The additive errors on them will lead to additive errors on τ_k, τ'_k, which will be detected in the next step.

Fig. 11. The verification protocol.

By letting each party play as a client and a server, the client-server model can be reduced to the standard MPC model. Thus, our protocol achieves malicious security against up to $n-1$ corruptions of n parties in the standard model.

The communication cost of $\varPi_{\mathsf{main}}$ is $(20W_I + 18W_O + 8W + 72G_A + 56G_X)n\lambda$ bits in the hybrid model. If we use $\varPi_{\mathsf{prep}}$ to realize $\mathcal{F}_{\mathsf{prep}}$, the execution of $\varPi_{\mathsf{prep}}$ requires communication of $(12nm_T + 16nm_B)\lambda = (24W_I + 24W_O + 16W + 48G_A + 24G_X)n\lambda$ bits, resulting in a total communication of $(44W_I + 42W_O + 24W + 120G_A + 80G_X)n\lambda$ bits. Since $W = G_A + G_X + W_I$, the total communication for the complete protocol is $(68W_I + 42W_O + 144G_A + 104G_X)n\lambda$ bits. Detailed security proof and cost analysis are provided in the full version of this paper.

7 Performance Evaluation

We now demonstrate the efficiency of our protocol. In Sect. 7.1, we provide a comparison of the concrete communication cost of our protocol $\varPi_{\mathsf{main}}$ with other state-of-the-art protocols. We also implement and benchmark the performance of our protocol, the results of which are given in Sect. 7.2.

Protocol Π_{Garbling}

Garbling Phase

Let λ be the computational security parameter and κ be the statistical security parameter. All the servers agree on the public parameter $pp \leftarrow \mathsf{Setup}(\lambda, \kappa)$ from the public-key encryption scheme $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ constructed in Section 4.

1. **Initialization.** Set $m_T = 4G_A + 2G_X + 2W_I + 2W_O$, $m_B = W$, $m_R = W + 1$. The servers send $(\mathsf{Init}, m_T, m_R, m_B)$ to $\mathcal{F}_{\mathsf{prep}}$ and receive the outputs from $\mathcal{F}_{\mathsf{prep}}$.
2. **Preparing Mask Sharings.** For each wire w, all the servers take one random sharing of a bit generated by **Random Bits** of $\mathcal{F}_{\mathsf{prep}}$ as $[\![\lambda_w]\!]$. λ_w serves as the mask of the wire value of w.
3. **Preparing Separate MAC Keys.** For each input and output wire w:
   (a) The servers take a triple generated by $\mathcal{F}_{\mathsf{prep}}$ as $[\![\Delta_w]\!], [\![\Delta'_w]\!], [\![\Delta_w \cdot \Delta'_w]\!]$.
   (b) The servers take another triple $[\![a_w]\!], [\![b_w]\!], [\![c_w]\!]$ and run Π_{Mult} on $[\![\Delta_w]\!]$ and $[\![\lambda_w]\!]$ to obtain $[\![\Delta_w \cdot \lambda_w]\!]$.
   (c) The servers run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check the MACs on all the opened values, i.e. $(W_I + W_O)$ pairs of d, e opened in $(W_I + W_O)$ executions of Π_{Mult}.
4. **Revealing Input Masks.** For each input wire attached to each client C_i:
   (a) Each server sends his shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ to C_i.
   (b) C_i reconstructs $\lambda_w, \Delta_w, \Delta'_w, \lambda_w \cdot \Delta_w, \Delta_w \cdot \Delta'_w$. If $\lambda_w \cdot \Delta_w$ is not equal to the product of λ_w and Δ_w, or $\Delta_w \cdot \Delta'_w$ is not equal to the product of Δ_w and Δ'_w, abort the protocol.
5. **Sending Input Wire Values.** For each input wire w attached to each client C_i, C_i sends $v_w \oplus \lambda_w$ to all the servers, where v_w is the wire value of w.
6. **Preparing Shares of Wire Labels.** For each wire w, the servers take a random sharing $[\![k_{w,0}]\!]$ generated by **Random Values** of $\mathcal{F}_{\mathsf{prep}}$. Let S_i's share of each $[k_{w,0}]$ be $k_{w,0}^{(i)}$.
7. **Sending Public Keys.**
   (a) For each wire w, the servers compute their shares $\{g^{k_{w,0}^{(i)}}\}_{i=1}^n$ of $g^{[k_{w,0}]}$ and send them to S_1. S_1 computes and sends $g^{k_{w,0}}$ to all servers.
   (b) The servers use a random sharing $[\![r]\!]$ generated by **Random Values** of $\mathcal{F}_{\mathsf{prep}}$. Then all servers compute their shares of $g^{[r]}$ and send them to S_1. S_1 computes and sends g^r to all the servers.
   (c) The servers call $\mathcal{F}_{\mathsf{Coin}}$ to get a random seed in $\mathbb{F}_p$ and expand it to get random values $\theta_1, \dots, \theta_W \in \mathbb{F}_p$, and locally compute $[\![\tau]\!] = \sum_{i=1}^W \theta_i \cdot [\![k_{w_i,0}]\!] + [\![r]\!]$. Then, the servers run Π_{Open} to open τ.
   (d) Each server checks whether $g^\tau = \prod_{i=1}^W (g^{k_{w_i,0}})^{\theta_i} \cdot g^r$. If not, abort the protocol.
   (e) All servers locally compute $g^{k_{w,1}} = g^{k_{w,0}} \cdot g^{\Delta}$ for each wire w.
8. **Garbling the Circuit.** For each gate g with input wires a, b and output wire c, let $f_g : \{0,1\}^2 \to \{0,1\}$ be the function computed by the gate.
   (a) **Computing Sharings of Output Labels.** All the servers jointly compute SPDZ sharings of
   $$\chi_1 = f_g(0 \oplus \lambda_a, 0 \oplus \lambda_b) \oplus \lambda_c, \qquad \chi_2 = f_g(0 \oplus \lambda_a, 1 \oplus \lambda_b) \oplus \lambda_c,$$
   $$\chi_3 = f_g(1 \oplus \lambda_a, 0 \oplus \lambda_b) \oplus \lambda_c, \qquad \chi_4 = f_g(1 \oplus \lambda_a, 1 \oplus \lambda_b) \oplus \lambda_c.$$
   Note that $\lambda_w^2 = \lambda_w$ for each wire w.
   - For AND gates, servers run Π_{Mult} to compute $[\![\lambda_a \cdot \lambda_b]\!], [\![\lambda_c \cdot \lambda_b]\!], [\![\lambda_a \cdot \lambda_c]\!], [\![\lambda_a \cdot \lambda_b \cdot \lambda_c]\!]$. Note that each χ_j can be viewed as a linear combination of $\{1, \lambda_a, \lambda_b, \lambda_c, \lambda_a \cdot \lambda_b, \lambda_a \cdot \lambda_c, \lambda_b \cdot \lambda_c, \lambda_a \cdot \lambda_b \cdot \lambda_c\}$.
   - For XOR gates, note that $\chi_1 = \chi_4 = \lambda_a \oplus \lambda_b \oplus \lambda_c$ and $\chi_2 = \chi_3 = 1 \oplus \lambda_a \oplus \lambda_b \oplus \lambda_c$. All servers run Π_{Mult} to compute $[\![\lambda_a \cdot \lambda_b]\!]$ and then locally compute $[\![\lambda_a \oplus \lambda_b]\!] = [\![\lambda_a]\!] + [\![\lambda_b]\!] - 2 \cdot [\![\lambda_a \cdot \lambda_b]\!]$. Similarly, they get $[\![\lambda_a \oplus \lambda_b \oplus \lambda_c]\!]$ using one call of Π_{Mult}.

   (b) **Verification of MACs.** The servers run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check the MACs on all the opened values, i.e. τ and $4G_A + 2G_X$ pairs of d, e opened in $4G_A + 2G_X$ executions of Π_{Mult}.
   (c) **Encrypting Output Labels.**
      i. All servers locally compute $[\![\chi_j]\!]$ and $[x_{c,j}] = [k_{c,0}] + [\chi_j \cdot \Delta]$, $j = 1, 2, 3, 4$.
      ii. Each server S_i encrypts his share $x_{c,1}^{(i)}$ (of $[x_{c,1}]$) by $\mathsf{Enc}(pp, g^{k_{a,0}}, g^{k_{b,0}}, x_{c,1}^{(i)})$, $x_{c,2}^{(i)}$ by $\mathsf{Enc}(pp, g^{k_{a,0}}, g^{k_{b,1}}, x_{c,2}^{(i)})$, $x_{c,3}^{(i)}$ by $\mathsf{Enc}(pp, g^{k_{a,1}}, g^{k_{b,0}}, x_{c,3}^{(i)})$, and $x_{c,4}^{(i)}$ by $\mathsf{Enc}(pp, g^{k_{a,1}}, g^{k_{b,1}}, x_{c,4}^{(i)})$. Then, S_i sends the ciphertexts to S_1.

Fig. 12. Protocol for the garbling phase.

Protocol Π_{Eval}

Circuit Evaluation Phase

1. **Revealing Input Labels.** For each input wire w, all the servers send their shares of $[k_{w,v_w \oplus \lambda_w}]$ to S_1. S_1 checks whether $k_{w,v_w \oplus \lambda_w}$ is consistent with the corresponding public key. If not, abort the protocol.
2. **Computing the Circuit.** S_1 computes the circuit gate by gate. For each gate with input wires a, b and output wire c, if S_1 knows $k_{a,v_a \oplus \lambda_a}, k_{b,v_b \oplus \lambda_b}$, he can use them to decrypt all the servers' shares of $k_{c,v_c \oplus \lambda_c}$. Then S_1 computes $g^{k_{c,v_c \oplus \lambda_c}}$ and compares it with $g^{k_{c,0}}$ and $g^{k_{c,1}}$ to learn $v_c \oplus \lambda_c$. If $g^{k_{c,v_c \oplus \lambda_c}}$ is not in $\{g^{k_{c,0}}, g^{k_{c,1}}\}$, abort the protocol.
3. **Sending Outputs.** For each output wire w attached to each client C_i:
   (a) S_1 sends $v_w \oplus \lambda_w$ and $k_{w,v_w \oplus \lambda_w}$ to C_i. Then C_i checks whether they match the public key $g^{k_{w,v_w \oplus \lambda_w}}$. If not, abort the protocol.
   (b) Each server sends his shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ to C_i.
   (c) C_i reconstructs $\lambda_w, \Delta_w, \Delta'_w, \lambda_w \cdot \Delta_w, \Delta_w \cdot \Delta'_w$. If $\lambda_w \cdot \Delta_w$ is not equal to the product of λ_w and Δ_w, or $\Delta_w \cdot \Delta'_w$ is not equal to the product of Δ_w and Δ'_w, abort the protocol.
   (d) C_i computes his output v_w from $v_w \oplus \lambda_w$ and λ_w.

Fig. 13. Protocol for the circuit evaluation phase.

7.1 Cost Analysis

Instantiation of [4] *via Le Mans* [36]. The LPN-based dishonest majority multiparty garbling protocol of [4], as proposed in their work, incurred a communication cost of $O(n \cdot \lambda)$ bits per gate *per party*. We observe that, by incorporating the preprocessing functionalities of Le Mans [36] as we did, and by requiring only one party to evaluate the garbled circuit, the cost of their protocol can be reduced to $O(\lambda)$ bits per gate per party. In order to ensure a fair comparison (unbiased by the underlying SPDZ functionalities used), we first calculate the communication cost per party of this upgraded version of their protocol. Using the results of Le Mans [36], each multiplication (respectively, opening) can be achieved with a communication cost of $12n$ (respectively, $2n$) field elements, and the upgraded protocol from [4] requires a total communication cost of $((12 + 48k + 104\ell) \cdot G_A + (2 + 2k) \cdot W_I + 2 \cdot W_O)\, n$ bits, where $k, \ell \in \mathsf{poly}(\lambda)$. For more details, we refer the readers to the full version of this paper.

Remark 2. The malicious variant of [36] is applicable only in large fields, and it cannot be applied directly to the binary field used in [4]. For simplicity, we omit this distinction in the comparison with [4].

Comparison of Concrete Costs. We next calculate and compare the concrete cost of communication per party between our protocol and [4], including both garbling and evaluation phases, on the AES-128 and SHA-256 circuits. The former has 6400 AND, 28176 XOR, and 2087 INV gates (where each INV(x) can

Table 1. Comparison of the communication cost per party (in MB) incurred in the secure computation of the AES-128 and SHA-256 circuits. The computational security parameter is set to $\lambda = 128$ for both protocols. The statistical security parameter for the protocol of [4] is set to $\kappa = 80$.

Circuit	Ben-Efraim et al. [4]	This work
AES-128	768.11	65.33
SHA-256	2709.04	239.66

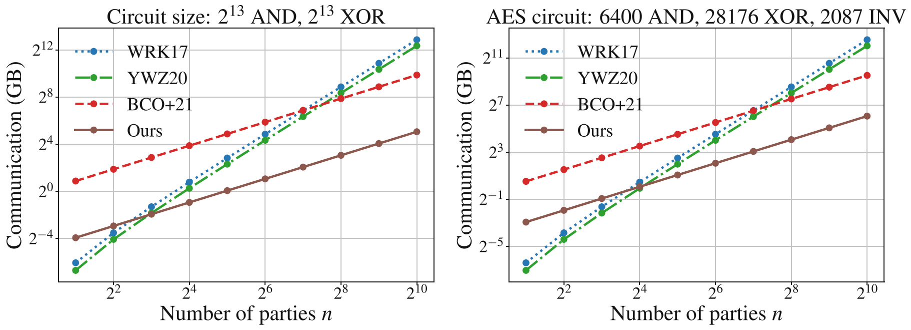


Fig. 14. The communication overhead of multiparty garbling protocols. The security parameters are set to $\lambda = 128, \kappa = 80$ in all cases.

be computed as $1 \oplus x$), while the latter has 22573 AND, 110644 XOR, and 1856 INV gates. The cost of communication per party can be found in Table 1. Considering statistical security parameter $\kappa = 80$ for the protocol of [4] (where they use s to denote it), we observe that our protocol achieves approximately 11.7$\times$ and 11.3$\times$ improvements in communication cost on the AES-128 and SHA-256 circuits, respectively.

We also compare with [3] which achieves $O(|C|)$ communication assuming a strong honest majority. We note that our protocol outperforms that of [3] in terms of a total cost of communication for up to $n \approx 3500$ parties (we set corruption threshold $t = \lfloor \frac{n-1}{4} \rfloor$ in their protocol for this comparison). This evidence further reinforces the practicality of our protocol.

Finally, we compare with Wang et al. [39] and Yang et al. [42] that require $O(|C|n^2)$ overall communication. In Fig. 14, we compare our end-to-end communication overhead with the works of Wang et al. [39], Yang et al. [42], and Ben-Efraim et al. [4]. The sizes of the circuits are shown above figures. In these two cases, our protocol always has less communication overhead compared to [4], and outperforms [39, 42] when there are more than 2^3 to 2^4 parties.

7.2 Implementation and Experiments

We implement and benchmark the performance of our garbled circuits protocol, with a focus on the garbling and evaluation phases, which are our main contributions. We assume a trusted dealer who realizes the functionality $\mathcal{F}_{\mathsf{prep}}$ and distributes the corresponding secret shares to parties in a preprocessing phase. All experiments are done in an Amazon EC2 c5.24xlarge server with 96 vCPUs and 192 GB RAM. We emulate different network conditions with respect to bandwidth and latency. We assume a latency of 2ms in the LAN setting and 60ms in the WAN setting. For each of these settings, we emulate 10Gbps, 1Gbps, and 100Mbps networks. The implementation is written in C++ and is based on EMP-toolkit [38]. The elliptic curve operations are instantiated by NIST curve P-256 [34] and we use the implementations from the OpenSSL library. In our scheme, the garbling of XOR and AND gates consists of similar operations and takes almost the same amount of running time. Hence, we only record the cost of AND gates in the following experiments.

We first demonstrate the performance of our garbling protocol in different network settings and show the results in Table 2, with the number of parties $n \in \{2, 4, 8, 16, 32\}$. We separately show the average time usage per gate for the garbling phase and the transmission of the garbled circuits, of which time time usage increases with a reduction of bandwidth or an increase in latency. Since our protocol has constant-round communication, the impact of network latency is limited. The garbling time in WAN is only 1.1× to 2× compared to the time in LAN. Our protocol is also communication-efficient. The garbling time in a 100Mbps network compared to that in a 10Gbps network is less than 3.3× in LAN and less than 1.8× in WAN.

Because of the DDH-based encryption scheme, the computational cost of our scheme is higher than schemes that are based purely on symmetric-key operations, though ours has much lower communication complexity. Thus, we explore how multi-threading can improve the speed of garbling. The results are shown in Table 3. In LAN and WAN settings with different bandwidths, we set up 12 parties and increase the number of threads from 1 to 8. In all test cases, the running time of the circuit garbling significantly decreases with the increase of threads, with improvement up to 2.6×. Note that the saving brought by the multi-threading implementation is not strictly linear in the number of threads because the garbling phase requires communication and interaction.

We also show the microbenchmark of the circuit garbling phase in Table 4. The protocol is split into four major components: the generation of labels (k_w, g^{k_w}), the computation of mask bits (λ_a, λ_b) and their products $(\lambda_a\lambda_b, \lambda_a\lambda_c, \lambda_b\lambda_c, \lambda_a\lambda_b\lambda_c)$, the computation of $\{\chi_i\}_{i=1}^4$, and the encryption of table entries. As demonstrated in the table, the main computational bottleneck is the garbling of table entries. Fortunately, this cost can be greatly reduced by multi-threading as shown in the previous Table 3. Note that only the first two operations in Table 4 involve total communication linear in the number of parties and the size of the circuit; hence, their time usage is impacted by network conditions.

Table 2. Performance of the garbling protocol in different network conditions. The two numbers in each cell (separated by a comma) represent the time per gate (in 10^{-6} seconds) used for garbling circuits and sending the garbled tables to the evaluator. Each party runs on a single thread.

n	LAN			WAN		
	10Gbps	1Gbps	100Mbps	10Gbps	1Gbps	100Mbps
2	549,1	552,5	597,51	573,15	589,18	619,60
4	588,2	595,15	729,155	645,56	643,54	784,180
8	615,6	639,37	988,362	783,106	776,125	1075,419
16	669,13	726,79	1470,799	1052,227	1016,271	1689,895
32	786,36	925,164	2597,1605	1612,470	1497,555	2964,1850

Table 3. Performance of the garbling phase using different numbers of threads. The numbers are the average time per gate (in 10^{-6} seconds). The number of parties is 12.

Threads	LAN			WAN		
	10Gbps	1Gbps	100Mbps	10Gbps	1Gbps	100Mbps
1	635	667	1206	938	949	1408
2	382	414	934	681	711	1171
4	314	323	831	539	599	1023
8	242	271	776	561	570	976

Finally, we study the performance of the circuit evaluation phase. As shown in the second row of Table 5, the evaluation complexity is linear to the number of parties n. Compared to the garbling phase in Table 2, it becomes a bottleneck when n is large. Hence, we apply multi-threading so that its running time does not grow dramatically while increasing the number of parties (third row). Moreover, notice that the other parties are idle when P_1 is evaluating garbled circuits. We study the amortized cost when parties execute several independent instances of our MPC protocol in parallel, and different parties act as evaluators in these instances. The last row shows that the amortized running time is almost constant when there are n parallel instances.

Table 4. Microbenchmark of the garbling phase. The numbers are the average time per gate (in 10^{-6} seconds). The number of parties is 8 and each party runs on a single thread.

Operations	LAN			WAN		
	10Gbps	1Gbps	100Mbps	10Gbps	1Gbps	100Mbps
Generate labels	45	52	196	153	132	257
Compute λs and products	57	73	282	119	125	307
Compute χs	17	17	17	16	17	17
Garble table entries	495	495	492	493	501	492

Table 5. The garbled circuits evaluation time (in 10^{-6} seconds) per gate. The evaluation involves only local operations. In the third row, the number of threads for Evaluator is the same as the number of Garbler. The last row shows the amortized time per gate per circuit when running n MPC instances in parallel and each party acts as Evaluator (using a single thread) in one of the instances. The only exception is when $n = 32$, we only run 16 instances because of the lack of CPU resources.

n	4	8	16	32
Single-Thread	377	735	1465	2886
Multi-Thread	172	221	249	390
Parallel	94	95	95	188

Acknowledgement. J. Li and Y. Song were supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61361136003.

R. Ostrovsky was supported in part by NSF grants CNS-2246355, CCF-2220450, US-Israel BSF grant 2022370, and by Sunday Group.

References

1. Baum, C., Cozzo, D., Smart, N.P.: Using topgear in overdrive: A more efficient zkpok for SPDZ. In: Paterson, K.G., Stebila, D. (eds.) Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11959, pp. 274–302. Springer (2019). https://doi.org/10.1007/978-3-030-38471-5_12
2. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols (extended abstract). In: Ortiz, H. (ed.) Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA. pp. 503–513. ACM (1990). https://doi.org/10.1145/100216.100287

3. Beck, G., Goel, A., Hegde, A., Jain, A., Jin, Z., Kaptchuk, G.: Scalable multiparty garbling. In: Meng, W., Jensen, C.D., Cremers, C., Kirda, E. (eds.) Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023. pp. 2158–2172. ACM (2023). https://doi.org/10.1145/3576915.3623132
4. Ben-Efraim, A., Cong, K., Omri, E., Orsini, E., Smart, N.P., Soria-Vazquez, E.: Large scale, actively secure computation from LPN and free-xor garbled circuits. In: Canteaut, A., Standaert, F. (eds.) Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part III. Lecture Notes in Computer Science, vol. 12698, pp. 33–63. Springer (2021). https://doi.org/10.1007/978-3-030-77883-5_2
5. Ben-Efraim, A., Lindell, Y., Omri, E.: Optimizing semi-honest secure multiparty computation for the internet. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 578–590. ACM (2016). https://doi.org/10.1145/2976749.2978347
6. Ben-Efraim, A., Lindell, Y., Omri, E.: Efficient scalable constant-round MPC via garbled circuits. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10625, pp. 471–498. Springer (2017), https://doi.org/10.1007/978-3-319-70697-9_17
7. Ben-Efraim, A., Nielsen, M., Omri, E.: Turbospeedz: Double your online spdz! improving SPDZ using function dependent preprocessing. In: Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M. (eds.) Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11464, pp. 530–549. Springer (2019). https://doi.org/10.1007/978-3-030-21568-2_26
8. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: Simon, J. (ed.) Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA. pp. 1–10. ACM (1988), https://doi.org/10.1145/62212.62213
9. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11694, pp. 489–518. Springer (2019). https://doi.org/10.1007/978-3-030-26954-8_16
10. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators from ring-lpn. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12171, pp. 387–416. Springer (2020). https://doi.org/10.1007/978-3-030-56880-1_14
11. Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptol. **13**(1), 143–202 (2000). https://doi.org/10.1007/S001459910006
12. Carter, L., Wegman, M.N.: Universal classes of hash functions. J. Comput. Syst. Sci. **18**(2), 143–154 (1979). https://doi.org/10.1016/0022-0000(79)90044-8

13. Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols (extended abstract). In: Simon, J. (ed.) Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA. pp. 11–19. ACM (1988). https://doi.org/10.1145/62212.62214
14. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3876, pp. 285–304. Springer (2006). https://doi.org/10.1007/11681878_15
15. Damgård, I., Ishai, Y.: Constant-round multiparty computation using a black-box pseudorandom generator. In: Shoup, V. (ed.) Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3621, pp. 378–394. Springer (2005). https://doi.org/10.1007/11535218_23
16. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8134, pp. 1–18. Springer (2013). https://doi.org/10.1007/978-3-642-40203-6_1
17. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7417, pp. 643–662. Springer (2012). https://doi.org/10.1007/978-3-642-32009-5_38
18. Elgamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory **31**(4), 469–472 (1985). https://doi.org/10.1109/TIT.1985.1057074
19. Escudero, D., Goyal, V., Polychroniadou, A., Song, Y., Weng, C.: Superpack: Dishonest majority mpc with constant online communication. In: Advances in Cryptology - EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part II. p. 220-250. Springer-Verlag, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-30617-4_8
20. Garg, R., Yang, K., Katz, J., Wang, X.: Scalable mixed-mode mpc. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 109–109. IEEE Computer Society, Los Alamitos, CA, USA (may 2024).https://doi.org/10.1109/SP54263.2024.00106
21. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A.V. (ed.) Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA. pp. 218–229. ACM (1987). https://doi.org/10.1145/28395.28420
22. Håstad, J., Impagliazzo, R., Levin, L.A., Luby, M.: A pseudorandom generator from any one-way function. SIAM J. Comput. **28**(4), 1364–1396 (1999). https://doi.org/10.1137/S0097539793244708

23. Hazay, C., Orsini, E., Scholl, P., Soria-Vazquez, E.: Concretely efficient large-scale MPC with active security (or, tinykeys for tinyot). In: Peyrin, T., Galbraith, S.D. (eds.) Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11274, pp. 86–117. Springer (2018). https://doi.org/10.1007/978-3-030-03332-3_4
24. Hazay, C., Orsini, E., Scholl, P., Soria-Vazquez, E.: Tinykeys: A new approach to efficient multi-party computation. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III. Lecture Notes in Computer Science, vol. 10993, pp. 3–33. Springer (2018). https://doi.org/10.1007/978-3-319-96878-0_1
25. Hazay, C., Scholl, P., Soria-Vazquez, E.: Low cost constant round MPC combining BMR and oblivious transfer. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10624, pp. 598–628. Springer (2017). https://doi.org/10.1007/978-3-319-70694-8_21
26. Keller, M., Orsini, E., Scholl, P.: MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 830–842. ACM (2016). https://doi.org/10.1145/2976749.2978357
27. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III. Lecture Notes in Computer Science, vol. 10822, pp. 158–189. Springer (2018). https://doi.org/10.1007/978-3-319-78372-7_6
28. Larraia, E., Orsini, E., Smart, N.P.: Dishonest majority multi-party computation for binary circuits. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8617, pp. 495–512. Springer (2014). https://doi.org/10.1007/978-3-662-44381-1_28
29. Lindell, Y., Pinkas, B., Smart, N.P., Yanai, A.: Efficient constant round multi-party computation combining BMR and SPDZ. In: Gennaro, R., Robshaw, M. (eds.) Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9216, pp. 319–338. Springer (2015). https://doi.org/10.1007/978-3-662-48000-7_16
30. Mordell, L.J.: On the rational resolutions of the indeterminate equations of the third and fourth degree. In: Proc. Cambridge Phil. Soc. vol. 21, pp. 179–192 (1922)
31. Nevelsteen, W., Preneel, B.: Software performance of universal hash functions. In: Stern, J. (ed.) Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding. Lecture Notes in Computer Science, vol. 1592, pp. 24–41. Springer (1999). https://doi.org/10.1007/3-540-48910-X_3

32. Nisan, N., Zuckerman, D.: Randomness is linear in space. J. Comput. Syst. Sci. **52**(1), 43–52 (1996). https://doi.org/10.1006/JCSS.1996.0004
33. Pietrzak, K., Sjödin, J.: Weak pseudorandom functions in minicrypt. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations. Lecture Notes in Computer Science, vol. 5126, pp. 423–436. Springer (2008). https://doi.org/10.1007/978-3-540-70583-3_35
34. PUB, F.: Digital signature standard (dss). Fips pub pp. 186–192 (2000)
35. Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In: Johnson, D.S. (ed.) Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA. pp. 73–85. ACM (1989), https://doi.org/10.1145/73007.73014
36. Rachuri, R., Scholl, P.: Le mans: Dynamic and fluid MPC for dishonest majority. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13507, pp. 719–749. Springer (2022), https://doi.org/10.1007/978-3-031-15802-5_25
37. Stinson, D.R.: Universal hashing and authentication codes. In: Feigenbaum, J. (ed.) Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings. Lecture Notes in Computer Science, vol. 576, pp. 74–85. Springer (1991). https://doi.org/10.1007/3-540-46766-1_5
38. Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit (2016)
39. Wang, X., Ranellucci, S., Katz, J.: Global-scale secure multiparty computation. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 39–56. ACM (2017). https://doi.org/10.1145/3133956.3133979
40. Weil, A.: L'arithmétique sur les courbes algébriques. Acta mathematica **52**, 281–315 (1929)
41. Weng, C., Yang, K., Katz, J., Wang, X.: Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In: 42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. pp. 1074–1091. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00056
42. Yang, K., Wang, X., Zhang, J.: More efficient MPC from improved triple generation and authenticated garbling. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020. pp. 1627–1646. ACM (2020). https://doi.org/10.1145/3372297.3417285

43. Yao, A.C.: Theory and applications of trapdoor functions (extended abstract). In: 23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982. pp. 80–91. IEEE Computer Society (1982), https://doi.org/10.1109/SFCS.1982.45
44. Yao, A.C.: How to generate and exchange secrets (extended abstract). In: 27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986. pp. 162–167. IEEE Computer Society (1986). https://doi.org/10.1109/SFCS.1986.25

Updatable Private Set Intersection Revisited: Extended Functionalities, Deletion, and Worst-Case Complexity

Saikrishna Badrinarayanan[1](✉), Peihan Miao[2], Xinyi Shi[2], Max Tromanhauser[2], and Ruida Zeng[2]

[1] LinkedIn, Seattle, USA
sbadrinarayanan@linkedin.com
[2] Brown University, Providence, USA
{peihan_miao,xinyi_shi,max_tromanhauser,ruida_zeng}@brown.edu

Abstract. Private set intersection (PSI) allows two mutually distrusting parties each holding a private set of elements, to learn the intersection of their sets without revealing anything beyond the intersection. Recent work (Badrinarayanan et al., PoPETS'22) initiates the study of updatable PSI (UPSI), which allows the two parties to compute PSI on a regular basis with sets that constantly get updated, where both the computation and communication complexity only grow with the size of the small updates and not the large entire sets. However, there are several limitations of their presented protocols. First, they can only be used to compute the plain PSI functionality and do not support extended functionalities such as PSI-Cardinality and PSI-Sum. Second, they only allow parties to add new elements to their existing set and do not support arbitrary deletion of elements. Finally, their addition-only protocols either require both parties to learn the output or only achieve low complexity in an amortized sense and incur linear worst-case complexity.

In this work, we address all the above limitations. In particular, we study UPSI with semi-honest security in both the addition-only and addition-deletion settings. We present new protocols for both settings that support plain PSI as well as extended functionalities including PSI-Cardinality and PSI-Sum, achieving one-sided output (which implies two-sided output). In the addition-only setting, we also present a protocol for a more general functionality Circuit-PSI that outputs secret shares of the intersection. All of our protocols have worst-case computation and communication complexity that only grow with the set updates instead of the entire sets (except for a polylogarithmic factor). We implement our new UPSI protocols and compare with the state-of-the-art protocols for PSI and extended functionalities. Our protocols compare favorably when the total set sizes are sufficiently large, the new updates are sufficiently small, or in networks with low bandwidth.

Keywords: Private Set Intersection · Secure Two-Party Computation · Oblivious Data Structure

1 Introduction

Private Set Intersection (PSI) enables two distrusting parties, each holding a private set of elements, to jointly compute the intersection of their sets without revealing anything other than the intersection itself. Despite its simple functionality, PSI and its related notions have found many real-world applications including online advertising measurement (deployed by Google Ads [6,35]), secure password breach alert (deployed by Google Chrome [8], Microsoft Edge [3], Apple iCloud Keychain [4], etc.), mobile private contact discovery (deployed by Signal [9,37]), privacy-preserving contact tracing in a global pandemic (jointly deployed by Google and Apple [5,17,56]). The last several decades have witnessed enormous progress towards realizing PSI efficiently using various techniques achieving both semi-honest and malicious security [18,20,23–26,31,39,45,47,51].

In many real-world applications such as aggregated ads measurement and privacy-preserving contact tracing, PSI is performed on a regular (e.g., daily) basis with *updated* sets, where the updates can be small when compared to the entire sets. However, most of the existing work requires the two parties to perform a fresh PSI protocol every time. A recent work by Badrinarayanan et al. [16] initiates the study of *updatable PSI (UPSI)*, which allows the two parties to compute set intersections for sets that regularly get updated. Their work presents protocols for updatable PSI where both the computation and communication complexity only grow with the size of the updates and are independent of the size of the entire sets (except for a logarithmic factor). As a result, these protocols are orders of magnitudes faster than a fresh PSI protocol, especially when the updates are significantly smaller than the entire sets. Nevertheless, there are several limitations with the protocols in [16].

- **Functionality:** All the protocols presented in [16] are restricted to the *plain* PSI functionality, crucially leveraging the fact that parties learn all the elements in the intersection. However, certain real-world applications require more refined PSI functionalities that do not reveal the entire intersection but instead only provide aggregated information about the intersection or enable restricted computation on the data in the intersection. As two specific examples that model many applications such as online advertising measurement, *PSI-Cardinality* allows two parties to jointly learn the cardinality (or size) of their set intersection; *PSI-Sum* allows two parties, where one party additionally holds a private integer value associated with each element in her set, to jointly compute the sum of the associated integer values for all the elements in the intersection (together with the cardinality of the intersection).
- **Addition-Only:** [16] mainly focuses on the addition-only setting, where both parties can only *add* new elements to their existing old sets, and do not support arbitrary *deletion* of elements from their sets. Note that they present a protocol for *UPSI with weak deletion*, which allows the parties to refresh their sets every t days, namely, they will add a set of elements to their sets every day, and delete elements that were added to their sets t days ago.

However, it does not support arbitrary deletion, and the daily computation and communication complexity additionally grows with t.

– **Tradeoffs of the Addition-Only Protocols:** [16] presents two protocols for addition-only UPSI, each with its own tradeoffs. In particular, one protocol crucially requires *both* parties to learn the output (namely, two-sided UPSI), which may not be applicable in certain applications such as password breach alert. The other protocol allows a single party to learn the output (namely, one-sided UPSI), but it only achieves low computation and communication complexity in an *amortized* sense over many days; the *worst-case* complexity can be as high as linear in the entire sets. Note that one-sided UPSI is a strictly stronger functionality in the semi-honest setting (as considered in [16]) since the output-receiving party can simply send the output to the other party so as to achieve two-sided UPSI.

1.1 Our Results

In this work, we address all the aforementioned limitations by presenting new UPSI protocols for extended functionalities, supporting both addition and deletion of elements, achieving one-sided output and low worst-case complexity in both computation and communication. All of our protocols are secure in the semi-honest model, hence one-sided UPSI is a stronger functionality. In the setting with both addition and deletion, we achieve a slightly more general functionality than PSI-Sum as defined in [35,41], where we do *not* reveal the cardinality of the intersection along with the sum.

Besides the functionalities of plain PSI, PSI-Cardinality, and PSI-Sum that we discussed above, we consider a more general functionality of Circuit-PSI [18, 20,44,51,54], where the two parties learn the cardinality of the intersection as well as an additive secret share of each element in it. This functionality allows the two parties to perform further computation over the shares afterwards.

Note that we only consider Circuit-PSI in the addition-only setting. The challenge in achieving Circuit-PSI with both addition and deletion is as follows. Intuitively speaking, when deleting elements from the intersection, the parties must learn which existing secret shares to delete from the intersection (unless the parties update their entire secret shared intersection, where the complexity grows with the entire sets, which is undesirable). Given that they know *when* a particular secret share (not the element itself) was added to the intersection, this essentially reveals more information than what the ideal functionality outputs. Crucially, note that in the case of plain PSI with addition and deletion, this is not a problem since the ideal functionality's output also reveals *when* a particular element was added and deleted; and in the case of PSI-Cardinality or PSI-Sum, parties only learn aggregated information and this challenge doesn't arise in the protocol design. We summarize our results in comparison with [16] in Table 1.

Experiments. We implement all our protocols and compare their performance with the state-of-the-art protocols for PSI and extended functionalites [20,51]. As our communication grows with the size of the update and not the entire input

Table 1. Summary of our results in comparison to [16], including functionality, one-sided or two-sided output, support of addition and deletion of elements, and computation and communication complexity. PSI-Sum† denotes the variant of PSI-Sum that does not reveal the cardinality. N denotes the size of the entire sets and N_d denotes the size of the d-th update. t denotes the number of updates when parties refresh their sets in UPSI with weak deletion. $O^*(\cdot)$ denotes amortized complexity. For UPSI with both addition and deletion, we present two variants, one allowing each element to be added and deleted at most once, and the other allowing arbitrary additions and deletions of the same element.

Protocol	Functionality	Output	Addition/Deletion	Comp. & Comm. Complexity	
[16, $\Pi_{\text{UPSI-add-two}}$]	PSI	Two-Sided	Addition-Only	$O(N_d)$	
[16, $\Pi_{\text{UPSI-add-one}}$]	PSI	One-Sided	Addition-Only	$O^*(N_d \cdot \log N)$	
$\Pi_{\text{UPSI-Add}_{\text{psi}}}$	PSI	One-Sided	Addition-Only	$O(N_d \cdot \log N)$	
Figure 5, $\Pi_{\text{UPSI-Add}_{\text{ca}}}$	PSI-Cardinality				
Figure 5, $\Pi_{\text{UPSI-Add}_{\text{sum}}}$	PSI-Sum				
Figure 5, $\Pi_{\text{UPSI-Add}_{\text{circuit}}}$	Circuit-PSI	Secret Shared			
[16, $\Pi_{\text{UPSI-del}}$]	PSI	Two-Sided	Weak Deletion	$O(N_d \cdot t)$	
Figure 10, $\Pi_{\text{UPSI-Del}_{\text{psi}}}$	PSI	One-Sided	Addition & Deletion	Single Deletion $O(N_d \cdot \log N)$	Arbitrary Deletion $O(N_d \cdot \log^2 N)$
Figure 10, $\Pi_{\text{UPSI-Del}_{\text{ca}}}$	PSI-Cardinality				
Figure 10, $\Pi_{\text{UPSI-Del}_{\text{sum}}}$	PSI-Sum†				

(except by a logarithmic factor), we demonstrate a significant improvement, up to orders of magnitude, when the input sets grow sufficiently large with smaller updates. Although our usage of public key operations dampens the asymptotic impact on computation, in realistic WAN settings, our protocols are able to outperform prior work in end-to-end running time. We also compare our new one-sided addition-only UPSI protocol with [16] and show significant improvement in worst-case complexity.

1.2 Technical Overview

We discuss the technical challenges and novelties in this work. We start with addition-only UPSI. Let X, Y denote the old sets of the two parties P_0, P_1 respectively, and let X_d, Y_d denote their new added sets on Day d. For simplicity, assume $|X| = |Y| = N$ and $|X_d| = |Y_d| = N_d$.[1] Recall that we are mostly interested in the scenario when the set updates are significantly smaller than the entire sets, namely $N \gg N_d$. The parties have already learned $I = X \cap Y$ of the old sets, and they would like to learn the updated intersection $I_d = (X \cup X_d) \cap (Y \cup Y_d)$. We focus on one-sided UPSI, where only P_0 learns the output.

Addition-Only UPSI with Extended Functionalities. Our starting point is the one-sided addition-only UPSI protocol in [16]. They observe that it suffices to learn the set difference $I_d \setminus I$ on each day, which, from P_0's perspective, can be split into two disjoint sets, $(X_d \cap (Y \cup Y_d))$ and $(X \cap Y_d)$. They then develop protocols to compute the two sets individually, with complexity growing only

[1] Our constructions work for two sets with different sizes as well, which we elaborate in Sect. 3 and Sect. 4.

with N_d and not N. To compute UPSI-Cardinality, we similarly split $|I_d \setminus I|$ into $|X_d \cap (Y \cup Y_d)|$ and $|X \cap Y_d|$, and compute them individually. Note that this is not sufficient since the individual cardinalities reveal more information than the ideal functionality, which we will fix later.

Computing $|X_d \cap (Y \cup Y_d)|$*:* We first briefly describe the approach in [16] to computing $X_d \cap (Y \cup Y_d)$. Their key idea is to let P_1 store an encrypted version of her set on P_0's side; on each day, she updates this encrypted dataset based only on her new input Y_d. Here, they require a data structure that allows P_1 to obliviously update the dataset and P_0 to obliviously query and compute on the dataset. [16] constructs such an oblivious data structure via a binary tree and uses additively homomorphic encryption to compute on encrypted data. By carefully re-crafting the homomorphic operations on the encrypted data in the oblivious data structure, we design a method that reveals only the number of elements that are matched between X_d and the encrypted dataset $(Y \cup Y_d)$. This enables P_0 to learn $|X_d \cap (Y \cup Y_d)|$.

Computing $|X \cap Y_d|$*:* We review the approach in [16] to computing $X \cap Y_d$, which leverages Diffie-Hellman-based PSI in [16]. Unfortunately, it does not extend to updatable cardinality. To address this challenge, our idea is to compute $|X \cap Y_d|$ symmetrically on P_1's side using the oblivious data structure. In particular, we let P_0 store an encrypted version of his set on P_1's side that supports efficient and oblivious updates and queries. This way we can efficiently allow P_1 to learn $|X \cap Y_d|$.

Computing the Sum with One-Sided Output: There are two issues with our current approach: first, individual cardinalities should *not* be revealed to the parties; second, P_1 should *not* learn anything about the output. At a high level, P_0 learns the cardinality $|X_d \cap (Y \cup Y_d)|$ by decrypting a set of (homomorphically evaluated) ciphertexts and counts the number of 0's in them. This happens similarly for P_1 to learn $|X \cap Y_d|$. To fix the first issue, we develop a method to combine the two sets of ciphertexts, re-randomize and shuffle all of them, and then decrypt them at the end. The number of 0's reveals only the sum of $|X_d \cap (Y \cup Y_d)|$ and $|X \cap Y_d|$, rather than individual values. To fix the second issue, we use a 2-out-of-2 threshold encryption scheme. The parties will jointly decrypt all the ciphertexts only after the random shuffling, and the decrypted results are revealed only to P_0. This protocol can be further extended to PSI-Sum and Circuit-PSI by attaching a payload to each element and further leveraging additive homomorphism.

Worst-Case Logarithmic Complexity. The above construction relies heavily on the oblivious data structure presented in [16]. A critical drawback of the data structure is that it only achieves logarithmic complexity in an *amortized* sense, namely the average complexity over many days is low. However, the *worst-case* complexity can be as high as linear in the entire sets. In this work, we construct a new oblivious data structure with worst-case logarithmic complexity.

Recall that in our UPSI construction, P_1 store an encrypted version of her set, maintained in an oblivious data structure, on P_0's side. There are two requirements on the data structure: first, for each new element y added to P_1's set, P_1

can update the encrypted dataset without leaking any information about y to P_0; second, for each new element x added to P_0's set, P_0 can locally identify a small set of encryptions in the P_1's set that are potential matches to x.

At a high level, our construction works as follows. The encrypted dataset is maintained in a binary tree structure. Each element x identifies a designated, (pseudo)random root-to-leaf path, computed by a pseudorandom function $F_{\mathsf{k}}(x)$ with k known to both parties. As P_1 updates the tree, she will maintain the invariant that each element y always appears along its designated path. This allows P_0 to query for potential matches by collecting all elements in the appropriate path (i.e., potential matches to x will be found in the path designated by $F_{\mathsf{k}}(x)$). However, when a new element y is added to P_1's set, directly updating the designated path of y in P_0's storage reveals information about y being added to the tree. *Therefore, we need a mechanism for P_1 to add y to its designated path in P_0's storage while hiding the path from P_0.* In [16], this is achieved through a series of operations that update an entire level of the tree each time, resulting in an amortized logarithmic complexity, while the worst-case complexity is linear (when P_1 updates the leaf level of the tree).

Our solution takes inspiration from the Path ORAM construction [55]. Instead of updating the designated path, P_1 picks a random path each time, and "pushes down" the elements along that path as much as possible. The access pattern of tree updates consist of random paths, hence are oblivious to P_0. Note that Path ORAM has an additional logarithmic factor from tree recursions due to limited registers. We can remove the tree recursions since we do not have this restriction in UPSI, leading to a single logarithmic factor. We refer to Sect. 3 for more details of our addition-only UPSI protocols.

Supporting Deletion. Our oblivious data structure is inspired by ORAM, but the manner in which ORAM handles deletion (or modification) of memory content does not work for us. In Path ORAM, whenever x is accessed (or modified), x will be re-allocated to a new, freshly sampled random designated path. However, as discussed above, the designated path of x in our construction is fixed and known to both parties.

Our key idea is to keep the fixed designated path for the element and attach a payload of $+1$ or -1 to indicate addition or deletion. Specifically, when y is deleted from P_1's set, instead of deleting it from the data structure, she will *add* another y to the data structure with a payload of -1 indicating deletion. In other words, when y is added *or* deleted from P_1's set, she will add a new pair of encryptions $(\mathsf{Enc}(y), \mathsf{Enc}(+1))$ or $(\mathsf{Enc}(y), \mathsf{Enc}(-1))$ to the designated path of y. Recall that we can update the tree by accessing a random path, hence the access pattern remains oblivious to P_0. When x is added to P_0's set, P_0 will still identify all the encrypted pairs on the designated path of x as potential matches. However, the crucial challenge is when y is not in the intersection, we need to further hide from P_0 whether y was never added to the dataset, or y was added and then deleted (namely, $(y, +1)$ and $(y, -1)$ cancel out). To achieve this, we design a special protocol that, for each pair, if the element is a match, then the parties obtain a secret share of its corresponding payload ($+1$ or -1); otherwise they obtain a secret share of 0. Finally, they add up all these secret shares where $+1$'s and -1's are canceled out, revealing whether x is in the intersection.

There are several other challenges that arise in handling deletions. For instance, we need to bound the maximum node size of the tree, especially when there are unlimited, repeated elements being added to the same path. If we restrict each element to being added and deleted at most once, the complexity remains the same as in the addition-only protocols. A more nuanced analysis shows that with unlimited additions and deletions, the complexity incurs only an additional logarithmic factor. Another challenge arises in plain UPSI, when P_0 removes x and P_1 adds $y = x$ on the same day. After these updates, x is not in the intersection, and it should be further hidden that it was added and then deleted from the intersection. We refer to Sect. 4 for more details of how to handle these challenges and the full description of our UPSI protocols with both addition and deletion.

1.3 Related Work

There has been a long line of work towards realizing PSI efficiently using various techniques including Diffie-Hellman-based [34,35,40], RSA-based [13,27], circuit-based [33,46–48], oblivious transfer (OT)-based [21,28,39,44,49], fully homomorphic encryption (FHE)-based [22,23,25], and vector oblivious linear evaluation (VOLE)-based [18,26,31,51,54] approaches, achieving both semi-honest and malicious security [18,22,24,42,45,51,53].

As discussed earlier, certain applications require PSI with extended functionalities that do not reveal the entire intersection but rather enable restricted computation on the elements in the intersection. PSI-Cardinality and PSI-Sum model many applications such as aggregated ads measurement [35,41] and privacy-preserving contact tracing [17,56]. More generally, Circuit PSI [18,20,33,47,51,54] enables the two parties to learn secret shares of the set intersection, which can be used to securely compute any function using generic secure two-party computation protocols [32,58]. However, all these approaches study PSI or PSI with extended functionalities in the standalone setting, which do not support small updates to the sets beyond running a fresh protocol after each update.

To the best of our knowledge, [16] is the first work that formalizes and studies PSI in the updatable setting, which we have extensively discussed above. Another related work is [10], which studies delegatable PSI with small updates. Specifically, they allow multiple clients to outsource their (encrypted) private sets and delegate PSI computation to a cloud server. Clients can perform efficient updates on their outsourced sets where the computation and communication only grow with their updates. However, both the computation and communication costs of computing PSI still grow with size of the entire sets, and their protocol crucially requires the existence of a server.

Concurrent and Independent Work. A concurrent and independent work by Agarwal et al. [11] constructs a semi-honest secure UPSI protocol that supports arbitrary addition and deletion of elements. Their construction, which builds

UPSI from a new variant of structured encryption (StE), achieves worst-case communication and computation complexity that grows linearly with the size of the updates and poly-logarithmically with the size of the entire sets. Their framework supports the plain PSI functionality with two-sided output, and focuses on feasibility. In contrast, our work additionally achieves the extended functionalities with one-sided output (which implies two-sided output), and demonstrates concrete efficiency.

2 Preliminaries

Notation. We use λ, κ to denote the computational and statistical security parameters, respectively. For an integer $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$. A 2-out-of-2 additive secret share of a value $x \in \mathbb{Z}_n$ is denoted as $([\![x]\!]_0, [\![x]\!]_1)$ where $[\![x]\!]_0 \xleftarrow{\$} \mathbb{Z}_n$ and $[\![x]\!]_0 + [\![x]\!]_1 = x \mod n$. PPT stands for probabilistic polynomial time. By $\overset{c}{\approx}$ we mean two distributions are computationally indistinguishable.

Additively Homomorphic Encryption. An additively homomorphic encryption scheme is a public-key encryption sccheme that consists of a tuple of PPT algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ over message space $\mathcal{M}$ with correctness, chosen-plaintext attack (CPA) security, and linear homomorphism.

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$: On input of the security parameter, output a public key pk and a secret key sk.
- $c \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m)$: On input of a public key pk and a message $m \in \mathcal{M}$, output a ciphertext c.
- $m/\bot \leftarrow \mathsf{Dec}_{\mathsf{sk}}(c)$: On input of a secret key sk and a ciphertext c, output a plaintext m or the symbol $\bot$.
- $\mathsf{Enc}_{\mathsf{pk}}(m_0+m_1) \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m_0) \oplus \mathsf{Enc}_{\mathsf{pk}}(m_1)$: On input two ciphertexts of m_0, m_1 encrypted under pk, output a ciphertext for their sum.
- $\mathsf{Enc}_{\mathsf{pk}}(m_0 \cdot m_1) \leftarrow m_0 \odot \mathsf{Enc}_{\mathsf{pk}}(m_1)$: On input a plaintext message m_0 and a ciphertext of m_1 encrypted under pk, output a ciphertext for their product.

Threshold Additively Homomorphic Encryption. A $(2,2)$-threshold additively homomorphic encryption scheme consists of a tuple of PPT algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{PartDec}, \mathsf{FullDec})$ over message space $\mathcal{M}$.

- $(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{KeyGen}(1^\lambda)$: On input of the security parameter, output a public key pk and a pair of secret key shares sk_0 and sk_1.
- $c \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m)$: On input of a public key pk and a message $m \in \mathcal{M}$, output a ciphertext c.
- $\hat{c} \leftarrow \mathsf{PartDec}_{\mathsf{sk}_b}(c)$: On input a secret key share sk_b (for $b \in \{0,1\}$) and a ciphertext c, output a partially decrypted ciphertext $\hat{c}$.
- $m/\bot \leftarrow \mathsf{FullDec}_{\mathsf{sk}_b}(\hat{c})$: On input a secret key share sk_b (for $b \in \{0,1\}$) and a partially decrypted ciphertext $\hat{c}$ by the other secret key sk_{1-b}, output a plaintext m or the symbol $\bot$.

The scheme satisfies correctness and CPA security even given a secret key share sk_b for $b \in \{0,1\}$. It also supports linear homomorphic operations $\oplus$ and $\odot$.

Re-randomization. A re-randomization algorithm $\tilde{c} \leftarrow \mathsf{ReRand}_{\mathsf{pk}}(c)$ homomorphically adds an independently generated encryption of zero to c, resulting in a ciphertext $\tilde{c}$ that is indistinguishable from a fresh ciphertext encrypting the same message as c. We implicitly assume that each homomorphic operation is followed by a re-randomization process. This is required in our protocols to ensure that the randomness of the final ciphertext is independent of the randomness used in the original ciphertexts. For the popular (threshold) additively homomorphic encryption schemes such as exponential El Gamal encryption [29] and Paillier encryption [43], a homomorphically evaluated ciphertext can be made statistically identical to a fresh ciphertext. We refer to [29,43] for formal definitions of correctness and CPA security.

3 Addition-Only UPSI

3.1 Definition

In this section, we formalize the ideal functionality and security definition for addition-only UPSI. Consider two parties P_0 and P_1 who wish to run PSI on a daily basis with updated sets. In the addition-only setting, they each hold a private set and add new elements to their respective sets each day. They want to jointly compute their set intersection (or extended functionalities) on their updated sets without revealing anything beyond that. We formalize addition-only UPSI as a special case of secure two-party computation with a reactive functionality defined in Fig. 1.

Initialization: $X = \emptyset$ and $Y = \emptyset$.

Day d**:**

- **Public Parameters**: The number of additions that P_0 and P_1 are performing: $|X_d|$ and $|Y_d|$, respectively.
- **Inputs:**
  P_0 inputs a set $X_d \subseteq \{0,1\}^*$ where $X_d \cap X = \emptyset$. In $\mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{sum}}}$, X_d includes an integer value associated with each set member (i.e., v_i is associated with $x_i \in X_d$).
  P_1 inputs a set $Y_d \subseteq \{0,1\}^*$ where $Y_d \cap Y = \emptyset$.
- **Update**: On receiving the inputs from both parties, the ideal functionality updates $X = X \cup X_d$ and $Y = Y \cup Y_d$.
- **Output:**
  In $\mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{psi}}}$, P_0 learns the intersection $I_d = X \cap Y$.
  In $\mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{ca}}}$, P_0 learns the cardinality of the intersection $C_d = |X \cap Y|$.
  In $\mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{sum}}}$, P_0 learns $C_d = |X \cap Y|$ and $V_d = \sum_{i:x_i \in X \cap Y} v_i$.
  In $\mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{circuit}}}$, both parties learn $C_d = |X \cap Y|$. For each new element z being added to the intersection, P_0 learns $[\![z]\!]_0$ and P_1 learns $[\![z]\!]_1$ as an additive secret share for z.

Fig. 1. Ideal functionalities for one-sided addition-only UPSI: $\mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{psi}}}$, $\mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{ca}}}$, $\mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{sum}}}$, $\mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{circuit}}}$.

Let $X_{[D]} = \{X_1, \ldots, X_D\}$ and $Y_{[D]} = \{Y_1, \ldots, Y_D\}$ be the inputs for P_0 and P_1 after D days, respectively. Let $\mathsf{View}_b^{\Pi,D}(X_{[D]}, Y_{[D]})$ and $\mathsf{Out}_b^{\Pi,D}(X_{[D]}, Y_{[D]})$ be the view and outputs of P_b (for $b \in \{0,1\}$) in the protocol Π at the end of D days, respectively. For a functionality $\mathcal{F}$, let $\mathcal{F}_b$ be the output for P_b in the D days. Note that $\mathcal{F}_1 = \perp$ in all the functionalities except for $\mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{circuit}}}$.

Definition 1 (One-Sided Addition-Only UPSI). *A protocol Π is semi-honest secure with respect to ideal functionality $\mathcal{F} \in \{\mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{psi}}}, \mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{ca}}}, \mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{sum}}}, \mathcal{F}_{\mathsf{UPSI\text{-}Add}_{\mathsf{circuit}}}\}$ if there exists PPT simulators Sim_0 and Sim_1 such that, for any $D \in \mathbb{N}^+$ and any inputs $(X_{[D]}, Y_{[D]})$,*

$$\left(\mathsf{View}_0^{\Pi,D}(X_{[D]}, Y_{[D]}), \mathsf{Out}_1^{\Pi,D}(X_{[D]}, Y_{[D]})\right) \overset{c}{\approx} \left(\mathsf{Sim}_0(1^\lambda, X_{[D]}, \mathcal{F}_0(X_{[D]}, Y_{[D]})), \mathcal{F}_1(X_{[D]}, Y_{[D]})\right),$$

$$\left(\mathsf{View}_1^{\Pi,D}(X_{[D]}, Y_{[D]}), \mathsf{Out}_0^{\Pi,D}(X_{[D]}, Y_{[D]})\right) \overset{c}{\approx} \left(\mathsf{Sim}_1(0^\lambda, Y_{[D]}, \mathcal{F}_1(X_{[D]}, Y_{[D]})), \mathcal{F}_0(X_{[D]}, Y_{[D]})\right).$$

Notation. Let $\Pi_{\mathsf{AHE}} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{PartDec}, \mathsf{FullDec})$ be a $(2,2)$-threshold additively homomorphic encryption scheme (see definition in Sect. 2) over plaintext space $\mathbb{Z}_q$ for a prime q. Without loss of generality we assume all the set elements are in $\mathbb{Z}_q$ (if not, we can apply a collision-resistant hash function $H : \{0,1\}^* \to \mathbb{Z}_q$ on all the elements and perform PSI on the hash outputs). Let $F : \{0,1\}^\lambda \times \mathbb{Z}_q \to \{0,1\}^\lambda$ be a pseudorandom function (PRF). For a bit string $s \in \{0,1\}^n$, let $s_{[1:i]}$ denote the prefix of s of length i (for $i \in [n]$).

Consider a binary tree data structure with tree height L and 2^L leaves, let $\ell \in \{0, 1, \ldots, 2^L - 1\}$ denote the ℓ-th leaf node of the tree. Any leaf node ℓ defines a unique path from the root to the leaf. We use $\mathcal{P}(\ell)$ to denote such a path, and $\mathcal{P}(\ell, k)$ to denote the node in $\mathcal{P}(\ell)$ at level k of the tree (for $k \in \{0, 1, \ldots, L\}$). Let σ denote the maximum tree node size and ρ denote the stash size of our oblivious data structure.

3.2 Construction

In this section, we present our addition-only UPSI protocols. As briefly discussed in Sect. 1.2, each party stores an encrypted version of its set on the other party's storage. We first describe our new oblivious data structure maintained in a binary tree.

Oblivious Data Structure. Say P_1 is the data owner, who stores her encrypted set on P_0's side. Initially, the binary tree is empty with depth 0. Each node of the tree has a maximum capacity of σ elements. As P_1 adds new elements to the tree, she will gradually increase the tree depth. Figure 2 illustrates a tree of depth 3. Each element x is associated with a designated path computed by $F_{\mathsf{k}}(x)$, where F is a pseudorandom function and k is a secret key known to both parties. When a new element x is added to P_1's set, P_1 will add x to the one of the nodes in the root-to-leaf path ending at leaf node $F_{\mathsf{k}}(x)$, but in an oblivious way. In the example in Fig. 2, the designated path of x is $F_{\mathsf{k}}(x) = 001$, and P_1 will obliviously add x to one of the four nodes on the red path. To do so, P_1 first adds x to the root node of the tree. Then she samples a random root-to-leaf path ℓ of the tree, and collects all the elements in that random path. For every element x^* in that random path (note that this includes x, because x was just added to the root), P_1 will "push down" x^* along the random path ℓ as much as possible subject to the constraint that x^* is still on its designated path $F_{\mathsf{k}}(x^*)$. In the example, $\ell = 011$, and P_1 considers all the elements on the blue path. She can push x down one level since it overlaps with the red path. For another element y, suppose $F_{\mathsf{k}}(y) = 011$, then P_1 can push it down to the leaf level. For the element z, suppose $F_{\mathsf{k}}(z) = 010$, then P_1 cannot push it down further. Note that this process is oblivious to P_0 since the access pattern for *any* element is a random path. In the example, the access pattern for x is a random path ℓ that is completely independent of x.

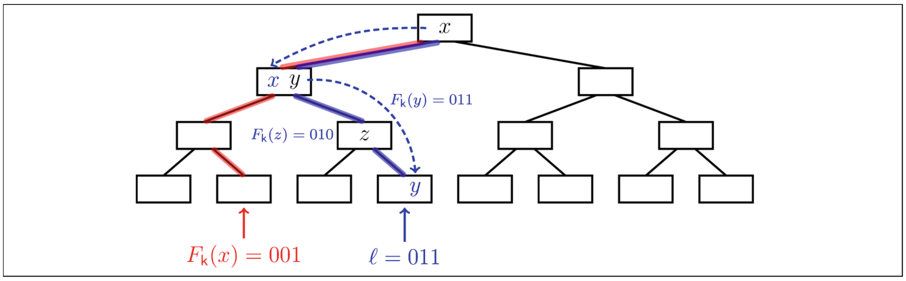

x
x y
$F_{\mathsf{k}}(y) = 011$
$F_{\mathsf{k}}(z) = 010$
z
y
$F_{\mathsf{k}}(x) = 001$
$\ell = 011$

Fig. 2. Illustration of adding an element x to a tree with depth 3. (Color figure online)

Some details were omitted in the above description for the sake of simplicity. First, when pushing down element along the random path ℓ, another constraint is that no node exceeds the maximum capacity of σ. Second, if there are extra elements that cannot fit into the maximum capacity of the random path, P_1 puts them into a *stash*, which has maximum capacity ρ. Both σ and ρ are defined as part of the security parameters of the protocol. We present this subroutine formally as UpdateTree in Fig. 3. This subroutine will also be used in our UPSI with both addition and deletion protocols, with slight modifications (highlighted in the figure). We discuss more details in Sect. 4.

Addition-Only UPSI-Cardinalty/Sum/Circuit-PSI. We now describe our new addition-only UPSI protocols (Fig. 5). P_0 maintains his elements $x \in X$ in an oblivious data structure consisting of a binary tree $\mathcal{D}_0$ and a stash $\mathcal{S}_0$. He stores an encrypted version of it on P_1's side, denoted as $(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0)$. Similarly, P_1 maintains her elements $y \in Y$ in an oblivious data structure $(\mathcal{D}_1, \mathcal{S}_1)$, and stores an encrypted version $(\widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1)$ on P_0's side. The encryption scheme is a (2, 2)-threshold additively homomorphic encryption. Recall from Sect. 1.2 that the set difference $I_d \setminus I$ on each day consists of two disjoint sets, $(X_d \cap Y)$ and $((X \cup X_d) \cap Y_d)$.

```
**Subroutine** UpdateTree($\{x_i\}_{i=1}^n, \{p_i\}_{i=1}^n, \mathcal{D}, \mathcal{S}, F_{\mathsf{k}}(\cdot), \mathsf{Enc}_{\mathsf{pk}}(\cdot)$):
1. Let $N$ be the total number of elements (excluding dummy ones) in the tree $\mathcal{D}$ and
   stash $\mathcal{S}$ after inserting $\{x_i\}_{i=1}^n$. Extend the tree depth to reach $L = \lceil \log_2 N \rceil$ if
   needed. Add empty nodes in the new levels of $\mathcal{D}$.
2. For each element and payload pair $(x_i, p_i)$ for $i \in [n]$:
   (a) Uniformly sample a random leaf node $\ell_i \xleftarrow{\$} \{0, 1, \ldots, 2^L - 1\}$ of the tree $\mathcal{D}$.
   (b) Remove all the elements from the path $\mathcal{P}(\ell_i)$ of the tree $\mathcal{D}$. Remove all the ele-
       ments from the stash $\mathcal{S}$. Combine all the removed elements (excluding dummy
       ones) with $(x_i, p_i)$ to get $\mathsf{path}_i$. In the UPSI with addition and deletion pro-
       tocols, if there are elements with opposite values, namely $(z, p)$ and $(z, -p)$,
       then remove both from $\mathsf{path}_i$.
   (c) For $k$ from $L$ down to 0:
       Consider the tree node $\mathcal{P}(\ell_i, k)$ at level $k$, remove up to $\sigma$ elements $(z, p)$ from
       $\mathsf{path}_i$ such that $\mathcal{P}(\ell_i, k) = \mathcal{P}(F_{\mathsf{k}}(z)_{[1:L]}, k)$, and add these elements to the node
       $\mathcal{P}(\ell_i, k)$ of $\mathcal{D}$.
   (d) Replace the stash $\mathcal{S}$ with all the elements left in $\mathsf{path}_i$. If there are more than
       $\rho$ elements left in $\mathsf{path}_i$, abort.
   (e) Pad every node in the path $\mathcal{P}(\ell_i)$ with dummy elements to reach a size of $\sigma$.
       Pad the stash $\mathcal{S}$ with dummy elements to reach a size of $\rho$.
3. For each $i \in [n]$, gather all the elements in the path $\mathcal{P}(\ell_i)$ and encrypt them to get
   $\widetilde{\mathsf{updates}}_i = \{(\mathsf{Enc}_{\mathsf{pk}}(x_j), \mathsf{Enc}_{\mathsf{pk}}(p_j))\}_{j=1}^{\sigma \cdot L}$. Encrypt all elements in the stash $\mathcal{S}$ to get
   $\widetilde{\mathcal{S}} = \{(\mathsf{Enc}_{\mathsf{pk}}(x_j), \mathsf{Enc}_{\mathsf{pk}}(p_j))\}_{j=1}^{\rho}$. Output $(\{(\widetilde{\mathsf{updates}}_i, \ell_i)\}_{i=1}^n, \widetilde{\mathcal{S}})$
```

Fig. 3. Subroutine UpdateTree that outputs a succinct update for the tree $\mathcal{D}$ that does not reveal the elements being added.

Let's first consider $(X_d \cap Y)$. Intuitively speaking, P_0 queries each $x_i \in X_d$ in the encrypted tree of Y, namely $(\widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1)$, to determine whether $x_i \in Y$. Specifically, for each $x_i \in X_d$, P_0 identifies a designated path $\ell = F_{\mathsf{k}}(x_i)$ and collects all the elements in the path ℓ from $\widetilde{\mathcal{D}}_1$, together with all the elements from $\widetilde{\mathcal{S}}_1$ (because x_i could potentially have been put there as well). These are all the candidate encryptions that could potentially match x_i. This process is presented formally as a subroutine GetPath in Fig. 4. To compute PSI-Cardinality,

```
**Subroutine** GetPath($\widetilde{\mathcal{D}}, \widetilde{\mathcal{S}}, F_{\mathsf{k}}(\cdot), x$):
1. Let $L$ be the height of the tree $\mathcal{D}$.
2. Compute the leaf node for the path containing $x$ as $\ell := F_{\mathsf{k}}(x)_{[1:L]}$.
3. Collect all the elements in the path $\mathcal{P}(\ell)$, combine them with the stash $\mathcal{S}$ to get
   $\widetilde{\mathsf{path}} = \{(\mathsf{Enc}(y_i), \mathsf{Enc}(p_i))\}_{i=1}^{\sigma \cdot L + \rho}$, and output $\widetilde{\mathsf{path}}$.
```

Fig. 4. Subroutine GetPath that outputs a collection of potential matching elements with x in the encrypted tree $\widetilde{\mathcal{D}}$ with stash $\widetilde{\mathcal{S}}$ organized according to the pseudorandom function F.

Initialization:

1. P_0 and P_1 jointly setup public and secret keys for a $(2,2)$-threshold additively homomorphic encryption scheme $(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{KeyGen}(1^\lambda)$ where P_0 receives $(\mathsf{pk}, \mathsf{sk}_0)$ and P_1 receives $(\mathsf{pk}, \mathsf{sk}_1)$. This can be done via a one-time secure two-party computation. The two parties agree on a randomly sampled PRF key $\mathsf{k} \xleftarrow{\$} \{0,1\}^\lambda$.
2. P_0 and P_1 generate initial trees with only an empty root and stash: $(\mathcal{D}_0, \mathcal{S}_0, \widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1)$ and $(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0, \mathcal{D}_1, \mathcal{S}_1)$, respectively.
3. Initialize $C_0 = 0$ in $\Pi_{\mathsf{UPSI\text{-}Add_{ca}}}$ and $\Pi_{\mathsf{UPSI\text{-}Add_{circuit}}}$, $C_0 = V_0 = 0$ in $\Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$.

Day *d*: P_0 and P_1 hold $(\mathcal{D}_0, \mathcal{S}_0, \widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1)$ and $(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0, \mathcal{D}_1, \mathcal{S}_1)$, respectively. Let L_0 be the tree height of $\mathcal{D}_0$ and $\widetilde{\mathcal{D}}_0$, and L_1 be the tree height of $\mathcal{D}_1$ and $\widetilde{\mathcal{D}}_1$. Both parties update L_0 and L_1 as they update the trees below. Let X, Y denote the two parties' sets at the end of the previous day, respectively. P_0 holds a new input set X_d and P_1 holds a new input set Y_d. Let $n = |X_d|$ and $m = |Y_d|$. In $\Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$, P_0 holds a value $v_i \in \mathbb{Z}_q$ associated with each element $x_i \in X_d$.

1. P_0 defines a payload for each element $x_i \in X_d$ depending on the functionality: $p_i = x_i$ in $\Pi_{\mathsf{UPSI\text{-}Add_{circuit}}}$, $p_i = v_i$ in $\Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$, and no payload is needed in $\Pi_{\mathsf{UPSI\text{-}Add_{ca}}}$.
2. $\boldsymbol{X_d}$ **tree update.** P_0 computes $m_1 = (\{(\widetilde{\mathsf{updates}}_i, \ell_i)\}_{i=1}^n, \widetilde{\mathcal{S}}_0') \leftarrow \mathsf{UpdateTree}(X_d, \{p_i\}_{i=1}^n, \mathcal{D}_0, \mathcal{S}_0, F_{\mathsf{k}}(\cdot), \mathsf{Enc}_{\mathsf{pk}}(\cdot))$, and sends it to P_1, who then replaces each path $\mathcal{P}(\ell_i)$ with $\widetilde{\mathsf{updates}}_i$ in $\widetilde{\mathcal{D}}_0$, and replaces $\widetilde{\mathcal{S}}_0$ with $\widetilde{\mathcal{S}}_0'$. Both parties update L_0 if needed.
3. **Candidates for $\boldsymbol{X_d \cap Y}$.** For each $x_i \in X_d$, P_0 computes $\{\mathsf{Enc}_{\mathsf{pk}}(y_{i,j})\}_{j=1}^{\sigma \cdot L_1 + \rho} \leftarrow \mathsf{GetPath}(\widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1, F_{\mathsf{k}}(\cdot), x_i)$, homomorphically subtracts x_i, and attaches an encryption of p_i to get $\widetilde{\mathsf{path}}_i = \{(\mathsf{Enc}_{\mathsf{pk}}(y_{i,j} - x_i), \mathsf{Enc}_{\mathsf{pk}}(p_i))\}_{j=1}^{\sigma \cdot L_1 + \rho}$. Then P_0 sends $m_2 = \{\widetilde{\mathsf{path}}_i\}_{i=1}^n$ to P_1.
4. **Candidates for $\boldsymbol{(X \cup X_d) \cap Y_d}$.** For each $y_j \in Y_d$, P_1 computes $\{(\mathsf{Enc}_{\mathsf{pk}}(x_{j,i}), \mathsf{Enc}_{\mathsf{pk}}(p_i))\}_{i=1}^{\sigma \cdot L_0 + \rho} \leftarrow \mathsf{GetPath}(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0, F_{\mathsf{k}}(\cdot), y_j)$, and homomorphically subtracts y_j to get $\widetilde{\mathsf{path}}_j = \{(\mathsf{Enc}_{\mathsf{pk}}(x_{j,i} - y_j), \mathsf{Enc}_{\mathsf{pk}}(p_i))\}_{i=1}^{\sigma \cdot L_0 + \rho}$.
5. **Combining candidates.** P_1 combines $\{\widetilde{\mathsf{path}}_j\}_{j=1}^m$ with $\{\widetilde{\mathsf{path}}_i\}_{i=1}^n$ received from P_0, randomly samples a mask $\alpha_k \xleftarrow{\$} \mathbb{Z}_q$ for each element in the combined set, and samples a random permutation π over $[\Gamma]$ where $\Gamma = \sigma \cdot (n \cdot L_1 + m \cdot L_0) + \rho \cdot (n + m)$. Compute and send the following to P_0:
$$m_3 = \pi\left(\{(\mathsf{PartDec}_{\mathsf{sk}_1}(\alpha_k \odot \mathsf{Enc}_{\mathsf{pk}}(a_k - b_k)), \mathsf{ReRand}_{\mathsf{pk}}(\mathsf{Enc}_{\mathsf{pk}}(p_k)))\}_{k=1}^{\Gamma}\right).$$
6. **Output generation.** P_0 fully decrypts the first element in each tuple of m_3 to get $\alpha_k(a_k - b_k)$. Let $K = \{k \mid \alpha_k(a_k - b_k) = 0\}$.
   - In $\Pi_{\mathsf{UPSI\text{-}Add_{ca}}}$, P_0 outputs $C_d = C_{d-1} + |K|$.
   - In $\Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$, P_0 computes $m_4 = \bigoplus_{k \in K} \mathsf{Enc}_{\mathsf{pk}}(p_k)$ and sends it to P_1. P_1 responds to P_0 with $m_4' = \mathsf{PartDec}_{\mathsf{sk}_1}(m_4)$. P_0 fully decrypts it to get $V = \mathsf{FullDec}_{\mathsf{sk}_0}(m_4')$, and outputs $V_d = V_{d-1} + V$.
   - In $\Pi_{\mathsf{UPSI\text{-}Add_{circuit}}}$, P_0 samples a random share $[\![z_k]\!]_0 \xleftarrow{\$} \mathbb{Z}_q$ for all $k \in K$, outputs $C_d = C_{d-1} + |K|$ and an updated share set with new random shares $\{[\![z_k]\!]_0\}_{k \in K}$. Additionally, P_0 computes and sends the following to P_1:
   $$m_4 = \{\mathsf{PartDec}_{\mathsf{sk}_0}(\mathsf{Enc}_{\mathsf{pk}}(p_k) \oplus \mathsf{Enc}_{\mathsf{pk}}(-[\![z_k]\!]_0))\}_{k \in K}.$$
   P_1 fully decrypts m_4 using sk_1 to get its shares $\{[\![z_k]\!]_1\}_{k \in K}$, and outputs $C_d = C_{d-1} + |K|$ and an updated share set with new random shares $\{[\![z_k]\!]_1\}_{k \in K}$.
7. $\boldsymbol{Y_d}$ **tree update.** P_1 computes $m_5 = (\{(\widetilde{\mathsf{updates}}_j, \ell_j)\}_{j=1}^m, \widetilde{\mathcal{S}}_1') \leftarrow \mathsf{UpdateTree}(Y_d, \perp, \mathcal{D}_1, \mathcal{S}_1, F_{\mathsf{k}}(\cdot), \mathsf{Enc}_{\mathsf{pk}}(\cdot))$, and sends it to P_0, who then replaces each path $\mathcal{P}(\ell_j)$ with $\widetilde{\mathsf{updates}}_j$ in $\widetilde{\mathcal{D}}_1$, and replaces $\widetilde{\mathcal{S}}_1$ with $\widetilde{\mathcal{S}}_1'$. Both parties update L_1 if needed.

Fig. 5. Protocols $\Pi_{\mathsf{UPSI\text{-}Add_{ca}}}, \Pi_{\mathsf{UPSI\text{-}Add_{sum}}}, \Pi_{\mathsf{UPSI\text{-}Add_{circuit}}}$ for one-sided addition-only UPSI functionalities $\mathcal{F}_{\mathsf{UPSI\text{-}Add_{ca}}}, \mathcal{F}_{\mathsf{UPSI\text{-}Add_{sum}}}, \mathcal{F}_{\mathsf{UPSI\text{-}Add_{circuit}}}$, respectively, with the differences among the three protocols highlighted.

P_0 homomorphically subtracts x_i from each candidate encryption, so it becomes an encryption of zero iff it is a match. This is presented as Step 3 in Fig. 5.

Symmetrically, for $((X \cup X_d) \cap Y_d)$, P_1 queries each $y_j \in Y_d$ in the encrypted tree of $(X \cup X_d)$, namely $(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0)$. Note that $(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0)$ needs to be first updated to contain X_d. In the protocol in Fig. 4, P_0 adds X_d to the oblivious data structure in Step 3. Then P_1 collects all the candidate encryptions for each $y_j \in Y_d$ and homomorphically subtracts y_j from them, as presented in Step 4.

In Step 5, P_1 combines all the candidate encryptions and homomorphically multiplies each one by a random scalar, so that a candidate encryption remains zero if it is a match, or random otherwise.[2] She then randomly shuffles all the candidate encryptions, partially decrypts them, and sends to P_0, who can then fully decrypt them and count the number of zeros.

Finally, P_1 adds Y_d to her oblivious data structure in Step 7. It is important to note that the order of tree updates for X_d and Y_d is critical in the protocol. In particular, the tree update for $(\widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1)$ can only occur after Step 3 to prevent doubly counting in PSI-Cardinality.

We can extend the protocol to PSI-Sum and Circuit-PSI by attaching a payload to each element and leveraging additive homomorphism on these payloads.

Addition-Only Plain UPSI. For addition-only plain UPSI $\mathcal{F}_{\mathsf{UPSI\text{-}Add_{psi}}}$, we don't have to store two trees. Instead, we can simply plug our new oblivious data structure into the addition-only UPSI protocol [16, $\Pi_{\mathsf{UPSI\text{-}add\text{-}one}}$] to achieve better concrete efficiency than the two-tree solution and much lower worst-case complexity than [16]. We present the protocol $\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$ in the full version of our paper [15].

3.3 Complexity, Correctness and Security

On each day d, let the entire set sizes of the two parties be N and M, respectively. Let the update set sizes be n and m, respectively. Then both the computation and communication complexity are $O(n \log M + m \log N)$, assuming σ and ρ are both $O(1)$. We state the theorem below and defer its proof to the full version of our paper [15].

Theorem 1. *Assuming Π is a secure $(2,2)$-threshold additively homomorphic encryption scheme, F is a pseudorandom function, the protocols $\Pi_{\mathsf{UPSI\text{-}Add_{ca}}}$, $\Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$, $\Pi_{\mathsf{UPSI\text{-}Add_{circuit}}}$ (Fig. 5) securely realize the ideal functionalities $\mathcal{F}_{\mathsf{UPSI\text{-}Add_{ca}}}$, $\mathcal{F}_{\mathsf{UPSI\text{-}Add_{sum}}}$, $\mathcal{F}_{\mathsf{UPSI\text{-}Add_{circuit}}}$ (Fig. 1), respectively, against semi-honest adversaries.*

4 UPSI with Addition and Deletion

4.1 Definition

Let $X_{[D]} = \{(X_1^+, X_1^-), \ldots, (X_D^+, X_D^-)\}$ and $Y_{[D]} = \{(Y_1^+, Y_1^-), \ldots, (Y_D^+, Y_D^-)\}$ be the inputs for P_0 and P_1 after D days, respectively. Here, X_d^+ denotes the

[2] Note that this holds because the plaintext space for the encryption scheme is $\mathbb{Z}_q$ for a prime q.

elements to be added to P_0's set on day d, and X_d^- denotes the elements to be deleted from P_0's set on day d; similarly, Y_d^+ and Y_d^- denote the elements to be added and deleted, respectively, for P_1 on day d. The ideal functionalities are defined in Fig. 6. Note that for $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{sum}}}$, we achieve a slightly more general functionality than PSI-Sum as defined in [35,41] (which is the definition used in our addition-only protocol) in that our functionality does *not* have to reveal the cardinality C_d along with V_d. Let $\mathcal{F}_0$ be the output for P_0 for all functionalities. Note that we don't consider the Circuit-PSI functionality in this setting, so P_1 has no output in the definition.

Initialization: $X = \emptyset$ and $Y = \emptyset$.

Day d:

- **Public Parameters**: For $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{psi}}}$, the number of additions and deletions performed each day: $|X_d^-|$, $|X_d^+|$, $|Y_d^-|$, $|Y_d^+|$.
  For $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{ca}}}$ and $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{sum}}}$, the *combined* number of additions and deletions performed each day: $|X_d^- \cup X_d^+|$ and $|Y_d^- \cup Y_d^+|$.
- **Inputs:**
  P_0 inputs an addition set $X_d^+ \subseteq \{0,1\}^*$ where $X_d^+ \cap X = \emptyset$ and a deletion set $X_d^- \subseteq X$. In $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{sum}}}$, X_d^+ includes a value associated with each set member (i.e., v_i is associated with $x_i \in X_d^+$).
  P_1 inputs an addition set $Y_d^+ \subseteq \{0,1\}^*$ where $Y_d^+ \cap Y = \emptyset$ and a deletion set $Y_d^- \subseteq Y$.
- **Update**: On receiving the inputs from both parties, the ideal functionality updates $X = (X \cup X_d^+) \setminus X_d^-$ and $Y = (Y \cup Y_d^+) \setminus Y_d^-$.
- **Output:**
  In $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{psi}}}$, P_0 learns the intersection $I_d = X \cap Y$.
  In $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{ca}}}$, P_0 learns the cardinality $C_d = |X \cap Y|$.
  In $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{sum}}}$, P_0 learns $V_d = \sum_{i:x_i \in X \cap Y} v_i$.

Fig. 6. Ideal functionalities for one-sided UPSI with both addition and deletion: $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{psi}}}$, $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{ca}}}$, and $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{sum}}}$.

Definition 2 (One-Sided UPSI with Addition and Deletion). *A protocol Π is semi-honest secure with respect to ideal functionality $\mathcal{F} \in \{\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{psi}}}, \mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{ca}}}, \mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{sum}}}\}$ if there exist PPT simulators Sim_0 and Sim_1 such that, for any $D \in \mathbb{N}^+$ and any inputs $(X_{[D]}, Y_{[D]})$,*

$$\left(\mathsf{View}_0^{\Pi,D}(X_{[D]}, Y_{[D]})\right) \overset{c}{\approx} \left(\mathsf{Sim}_0(1^\lambda, X_{[D]}, \mathcal{F}_0(X_{[D]}, Y_{[D]}))\right),$$

$$\left(\mathsf{View}_1^{\Pi,D}(X_{[D]}, Y_{[D]}), \mathsf{Out}_0^{\Pi,D}(X_{[D]}, Y_{[D]})\right) \overset{c}{\approx} \left(\mathsf{Sim}_1(1^\lambda, Y_{[D]}), \mathcal{F}_0(X_{[D]}, Y_{[D]})\right).$$

Notation. We use the same notation as in Sect. 3, except that instead of a (2, 2)-threshold additively homomorphic encryption scheme, we use a plain additively homomorphic encryption scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ (see definition in Sect. 2) over plaintext space $\mathbb{Z}_q$.

4.2 Construction

In this section, we present our UPSI protocols with both addition and deletion. The oblivious data structure presented in Sect. 3.2 only supports adding new elements to the tree. We first discuss how to extend the construction to also allow for deletion of elements from the tree.

Oblivious Data Structure with Deletion. Recall that each element x is associated with a designated path $F_{\mathsf{k}}(x)$. When P_1 adds a new element x to the tree, she will first add x to the root node of the tree. Then she samples a random path of the tree and pushes down elements along that random path as much as possible. To support deletion, P_1 first attaches a payload p to each element x. When x is added to P_1's set, she sets $p = +1$; when x is deleted from her set, she sets $p = -1$. Whenever an element x is added *or* deleted from her set, P_1 adds a new pair (x, p) to the tree following the exact same approach as described in $\mathsf{UpdateTree}$ (Fig. 3). The only minor difference is that when pushing down elements along the random path, if both $(x, +1)$ and $(x, -1)$ appear in that path, P_1 removes both of them from the tree.

This modified $\mathsf{UpdateTree}$ process remains oblivious to P_0 because the access pattern for addition or deletion of elements continues to be a random path together with the stash. Note that since additions and deletions of the same element have the same designated path, there is a higher probability of stash overflow if we use the same parameters of maximum node capacity σ and maximum stash capacity ρ as in the addition-only setting, hence we need to increase both parameters for our new protocols. We discuss the parameter implications in the security proofs in the full version of our paper [15].

Computation on Encrypted Tree. To compute on the encrypted tree, we take a different approach from the addition-only protocols. When P_0 queries an element x in the encrypted tree of Y, namely $(\widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1)$, he can still identify the designated path $\ell = F_{\mathsf{k}}(x)$ and collect all the candidate encryptions using $\mathsf{GetPath}$ (Fig. 4). However, there could be both $(\mathsf{Enc}(x), \mathsf{Enc}(+1))$ and $(\mathsf{Enc}(x), \mathsf{Enc}(-1))$ among these candidates. In case x was added and then deleted from tree, it should be indistinguishable to P_0 from the case where x was never added to the tree. We construct a subprotocol $\Pi_{\mathsf{CombinePath}}$ (Fig. 7) for the two parties to jointly learn a secret share of whether x is in the path, namely the sum of the associated payloads p for all the $(\mathsf{Enc}(x), \mathsf{Enc}(p))$ pairs.

Specifically, for each candidate encryption $(\mathsf{Enc}(y_i), \mathsf{Enc}(p_i))$, P_0 first homomorphically computes $\mathsf{Enc}(y_i - x + \alpha_i)$ for a randomly sampled α_i and sends it to P_1, which can then be decrypted by P_1 to γ_i. Note that $\alpha_i = \gamma_i$ iff $y_i = x$. Next, our goal is to design a special equality testing protocol such that if $\alpha_i = \gamma_i$ (i.e., $y_i = x$), then the two parties obtain a secret share of p_i, otherwise they obtain a secret share of 0. To do so, P_0 homomorphically computes two ciphertexts $m_{i,0} = \mathsf{Enc}(p_i - \beta_i)$ and $m_{i,0} = \mathsf{Enc}(-\beta_i)$ for a randomly sampled β_i. Then the two parties invoke a special secure two-party computation protocol with functionality $\mathcal{F}_{\mathsf{lookup}}$ (Fig. 8). The functionality $\mathcal{F}_{\mathsf{lookup}}$ takes $(\alpha_i, m_{i,0}, m_{i,1})$ from P_0 and γ_i from P_1 as input. If $\alpha_i = \gamma_i$, then $\mathcal{F}_{\mathsf{lookup}}$ outputs $m_{i,0}$ to P_1; otherwise it

Subprotocol $\Pi_{\mathsf{CombinePath}}((x, p, \widetilde{\mathsf{path}}), \mathsf{sk})$

Public Parameters: a public key pk for the additively homomorphic encryption scheme Π, and k as the number of pairs in $\widetilde{\mathsf{path}}$.

Inputs: An Initiator inputs an element x, an associated payload p, and a potential matching elements in an encrypted collection $\widetilde{\mathsf{path}} = \{(\mathsf{Enc}_{\mathsf{pk}}(y_i), \mathsf{Enc}_{\mathsf{pk}}(q_i))\}_{i=1}^{k}$. A Responder inputs the secret key sk corresponding to pk.

Output: Initiator and Responder receive a secret share of $\sum_{i\in[k]:x=y_i}(p \cdot q_i)$ over $\mathbb{Z}_q$.

1. For each $i \in [k]$, Initiator samples random masks $\alpha_i, \beta_i \xleftarrow{\$} \mathbb{Z}_q$ and homomorphically computes the following:
$$\begin{aligned} \mathsf{req}_i &= (\mathsf{Enc}_{\mathsf{pk}}(y_i) \oplus \mathsf{Enc}_{\mathsf{pk}}(\alpha_i - x)) \\ m_{i,0} &= p \odot \mathsf{Enc}_{\mathsf{pk}}(q_i) \oplus \mathsf{Enc}_{\mathsf{pk}}(-\beta_i) \\ m_{i,1} &= \mathsf{Enc}_{\mathsf{pk}}(-\beta_i) \end{aligned}$$
2. Initiator sends the request set $\{\mathsf{req}_i\}_{i=1}^{k}$ to Responder.
3. Responder decrypts each request with sk to get $\{\gamma_i\}_{i=1}^{k}$.
4. For all $i \in [k]$, both parties invoke $\mathcal{F}_{\mathsf{lookup}}$, where Initiator inputs $(\alpha_i, m_{i,0}, m_{i,1})$ as Sender and Responder inputs γ_i as Receiver, from which Responder receives m_i. Responder then sets $[\![r_i]\!]_1 = \mathsf{Dec}_{\mathsf{sk}}(m_i)$. Initiator sets $[\![r_i]\!]_0 = \beta_i$.
5. Each party P_b ($b \in \{0, 1\}$) outputs $\sum_{i=1}^{k}[\![r_i]\!]_b$.

Fig. 7. Subprotocol $\Pi_{\mathsf{CombinePath}}$ required for UPSI with addition and deletion.

Inputs: A Sender inputs (a, m_0, m_1) where $a \in \mathbb{Z}_q$ and (m_0, m_1) are two messages of equal length. A Receiver inputs $b \in \mathbb{Z}_q$.

Output: If $a = b$, then output m_0 to Receiver; otherwise output m_1 to Receiver.

Fig. 8. Ideal functionality $\mathcal{F}_{\mathsf{lookup}}$ required for the subprotocol $\Pi_{\mathsf{CombinePath}}$.

outputs $m_{i,1}$ to P_1. Therefore, if $\alpha_i = \gamma_i$, then P_1 obtains $\mathsf{Enc}(p_i - \beta_i)$, which can be decrypted to $p_i - \beta_i$, thereby forming a secret share of p_i with the other share β_i held by P_0. If $\alpha_i \neq \gamma_i$, then P_1 obtains a $\mathsf{Enc}(-\beta_i)$, which can be decrypted to $-\beta_i$, forming a secret share of 0 with P_0's share β_i. As a result, the two parties obtain a secret share of p_i if $y_i = x$, or a secret share of 0 otherwise. Finally, the two parties sum up all the secret shares to obtain a secret share of $\sum_{y_i=x} p_i$.

We present our subprotocol $\Pi_{\mathsf{CombinePath}}$ in Fig. 7 and defer its correctness and security proofs to the full version of our paper [15]. The functionality $\mathcal{F}_{\mathsf{lookup}}$ can be instantiated with a generic secure two-party computation protocol [32,58]. We present a more efficient realization utilizing oblivious transfer (OT) and the efficient OT extension [12,36] in Sect. 5.

UPSI-Cardinalty/Sum with Addition and Deletion. Next, we describe our new UPSI protocols with both addition and deletion for PSI-Cardinality and PSI-Sum, presented in Fig. 9. To compute PSI-Cardinality, we follow the similar framework as in the addition-only protocols (Fig. 5).

Initialization:

1. P_0 and P_1 independently generate key pairs for an additive homomorphic encryption scheme $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and share the public keys. Both parties agree on a randomly sampled PRF key $\mathsf{k} \xleftarrow{\$} \{0,1\}^\lambda$.
2. P_0 and P_1 generate initial trees with only an empty root and stash: $(\mathcal{D}_0, \mathcal{S}_0, \widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1)$ and $(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0, \mathcal{D}_1, \mathcal{S}_1)$, respectively. Initialize $\mathsf{Out}_0 = 0$.

Day $\boldsymbol{d}$: P_0 and P_1 hold $(\mathcal{D}_0, \mathcal{S}_0, \widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1)$ and $(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0, \mathcal{D}_1, \mathcal{S}_1)$, respectively. Let L_0 and L_1 be the heights of $\mathcal{D}_0$ (and $\widetilde{\mathcal{D}}_0$), and $\mathcal{D}_1$ (and $\widetilde{\mathcal{D}}_1$) respectively. Both parties update L_0 and L_1 as they update the trees below. Let X, Y denote the two parties' sets at the end of the previous day.
P_0 and P_1 have new input sets X_d^+, Y_d^+ which include elements they are adding to their set and X_d^-, Y_d^- of elements they are deleting. Denote $n = |X_d^+ \cup X_d^-|$, $m = |Y_d^+ \cup Y_d^-|$.
In $\mathit{\Pi}_{\mathsf{UPSI\text{-}Del_{sum}}}$, P_0 holds a value $v_i \in \mathbb{Z}_q$ associated with each element $x_i \in X_d^+ \cup X_d^-$.

1. P_0 defines a payload for each element $x_i \in X_d^+ \cup X_d^-$ depending on the functionality:
$$p_i := \begin{cases} (-1)^{(x_i \in X_d^-)} & \text{for } \mathit{\Pi}_{\mathsf{UPSI\text{-}Del_{ca}}} \\ (-1)^{(x_i \in X_d^-)} \cdot v_i & \text{for } \mathit{\Pi}_{\mathsf{UPSI\text{-}Del_{sum}}} \end{cases}$$
P_1 defines a payload for each element $y_i \in Y_d^+ \cup Y_d^-$: $q_j := (-1)^{(y_j \in Y_d^-)}$.
2. $\boldsymbol{X_d^+ \cup X_d^-}$ **tree update.** P_0 sends $(\{(\widetilde{\mathsf{updates}}_i, \ell_i)\}_{i=1}^n, \widetilde{\mathcal{S}}_0') \leftarrow \mathsf{UpdateTree}(X_d^+ \cup \widetilde{X_d^-, \{p_i\}}_{i=1}^n, \mathcal{D}_0, \mathcal{S}_0, F_\mathsf{k}(\cdot), \mathsf{Enc}_{\mathsf{pk}_0}(\cdot))$ to P_1. P_1 replaces each path $\mathcal{P}(\ell_i)$ with $\widetilde{\mathsf{updates}}_i$ in $\widetilde{\mathcal{D}}_0$, and replaces $\widetilde{\mathcal{S}}_0$ with $\widetilde{\mathcal{S}}_0'$.
3. **Secret shares for new elements of $\boldsymbol{X}$.** For all $x_i \in (X_d^+ \cup X_d^-)$, run $\mathit{\Pi}_{\mathsf{CombinePath}}$ with P_0 as $\mathsf{Initiator}$ inputting $(x_i, p_i, \widetilde{\mathsf{path}}_i \leftarrow \mathsf{GetPath}(\widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1, F_\mathsf{k}(\cdot), x_i), \mathsf{pk}_1)$ and P_1 as $\mathsf{Responder}$ inputting sk_1 corresponding to pk_1. They receive secret shares $[\![z_{x,i}]\!]_0$ and $[\![z_{x,i}]\!]_1$, respectively.
4. **Secret shares for new elements of $\boldsymbol{Y}$.** For all $y_j \in (Y_d^+ \cup Y_d^-)$, run $\mathit{\Pi}_{\mathsf{CombinePath}}$ with P_0 as $\mathsf{Responder}$ inputting sk_0 corresponding to pk_0) and P_1 as $\mathsf{Initiator}$ inputting $(y_j, q_j, \mathsf{path}_j \leftarrow \mathsf{GetPath}(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0, F_\mathsf{k}(\cdot), y_j), \mathsf{pk}_0)$. They receive secret shares $[\![z_{y,j}]\!]_0$ and $[\![z_{y,j}]\!]_1$, respectively.
5. $\boldsymbol{Y_d^+ \cup Y_d^-}$ **tree update.** P_1 sends $(\{(\widetilde{\mathsf{updates}}_j, \ell_j)\}_{j=1}^m, \widetilde{\mathcal{S}}_1') \leftarrow \mathsf{UpdateTree}(Y_d^+ \cup \widetilde{Y_d^-, \{q_j\}}_{j=1}^m, \mathcal{D}_1, \mathcal{S}_1, F_\mathsf{k}(\cdot), \mathsf{Enc}_{\mathsf{pk}_1}(\cdot))$ to P_0. P_0 replaces each path $\mathcal{P}(\ell_j)$ with $\widetilde{\mathsf{updates}}_j$ in $\widetilde{\mathcal{D}}_1$, and replaces $\widetilde{\mathcal{S}}_1$ with $\widetilde{\mathcal{S}}_1'$.
6. **Combine all the shares.** For $b \in \{0,1\}$, P_b computes $[\![z_d]\!]_b := \sum_{i=1}^n [\![z_{x,i}]\!]_b + \sum_{j=1}^m [\![z_{y,j}]\!]_b$.
7. **Output generation:** P_1 sends $[\![z_d]\!]_1$ to P_0, who then computes $\mathsf{Out}_d := \mathsf{Out}_{d-1} + [\![z_d]\!]_0 + [\![z_d]\!]_1$.
P_0 outputs Out_d for both $\mathit{\Pi}_{\mathsf{UPSI\text{-}Del_{ca}}}$ and $\mathit{\Pi}_{\mathsf{UPSI\text{-}Del_{sum}}}$.

Fig. 9. Protocols $\mathit{\Pi}_{\mathsf{UPSI\text{-}Del_{ca}}}$ and $\mathit{\Pi}_{\mathsf{UPSI\text{-}Del_{sum}}}$ for one-side UPSI with both addition and deletion functionalities $\mathcal{F}_{\mathsf{UPSI\text{-}Del_{ca}}}$ and $\mathit{\Pi}_{\mathsf{UPSI\text{-}Del_{sum}}}$, respectively, with differences between the two protocols highlighted.

In Step 1, if the element x_i is deleted from the set, the payload p_i should be -1 for $\mathit{\Pi}_{\mathsf{UPSI\text{-}Del_{ca}}}$, and $-v_i$ for $\mathit{\Pi}_{\mathsf{UPSI\text{-}Del_{sum}}}$. If the element x_i is added to the set, the payload p_i should be $+1$ for $\Pi_{\mathsf{UPSI\text{-}Del_{ca}}}$, and v_i for $\Pi_{\mathsf{UPSI\text{-}Del_{sum}}}$. In Step 2, P_0 adds all the elements in $X_d^+ \cup X_d^-$ to his tree using the oblivious data structure with deletion. In Step 3, P_0 queries each element $x_i \in X_d^+ \cup X_d^-$ in the encrypted tree of Y. For an element $x_i \in X_d^+$ to be added to the set, the two parties run $\Pi_{\mathsf{CombinePath}}$ to get a secret share of whether $x_i \in Y$. For an element $x_i \in X_d^-$ to be deleted from the set, they need to slightly modify $\Pi_{\mathsf{CombinePath}}$ to get a secret share of $(-1) \cdot$ (whether $x_i \in Y$). This means x_i was in the intersection but deleted from P_0's set in this step, so PSI-Cardinality is decreased by 1. In our protocol for $\Pi_{\mathsf{CombinePath}}$ (Fig. 7), P_0 inputs an additional value ($+1$ or -1) to be multiplied with the result, which is done homomorphically in the protocol. Symmetrically, P_1 queries each element $y_j \in X_d^+ \cup X_d^-$ in the encrypted tree of $(X \cup X_d^+) \setminus X_d^-$ in Step 4. After this, P_1 adds all the elements in $Y_d^+ \cup Y_d^-$ to her tree in Step 5 (recall that it must occur after Step 3).

Finally, the two parties add up all the secret shares in Step 6 and reveal the output in Step 7. This protocol can be naturally extended to PSI-Sum if P_0 attaches payloads of value $+v_i$ or $-v_i$ for each element x_i in $\mathsf{UpdateTree}$ and $\Pi_{\mathsf{CombinePath}}$. It is worth noting that parties only aggregate their secret shares at the end of the protocol, hence our PSI-Sum protocol does *not* have to reveal the cardinality of the intersection, which may be useful in certain applications.

Plain UPSI with Addition and Deletion. Interestingly, achieving plain UPSI is more challenging than PSI-Cardinality and PSI-Sum with addition and deletion. As briefly discussed in Sect. 1.2, one issue comes from the scenario when an element x is added by one party while being deleted by the other party on the same day. In our UPSI-Cardinality/Sum protocols, while adding and deleting x from the intersection both occur on the same day, their effect on the output cancels out when their secret shares are combined. However, in plain UPSI, parties need to learn the exact elements to be added or deleted. Revealing that x was first added and then deleted from the intersection on the same day discloses more information than the ideal functionality.

To address this issue, we carefully arranged the sequence of the addition and deletion operations, as presented in Fig. 10, such that deletions are dealt with in Step 1 before additions in Step 2. In other words, if x is deleted by P_0 while being added by P_1 on the same day, it will be first deleted from P_0's tree, so that it won't appear in the intersection when P_1 queries x in the encrypted tree. Since additions and deletions are done separately, both parties need to know $|X_d^-|$, $|X_d^+|$, $|Y_d^-|$, $|Y_d^+|$ on each day. This is different from UPSI-Cardinality/Sum where they only know $|X_d^- \cup X_d^+|$ and $|Y_d^- \cup Y_d^+|$, as reflected in the ideal functionalities (Fig. 6).

Furthermore, unlike UPSI-Cardinality/Sum where parties sum up all the secret shared results at the end of the protocol, they need to learn the results for each individual element in plain UPSI. However, they cannot reveal directly these results because doing so may disclose more information than the ideal functionality. Specifically, if an element x is deleted from both sets on the same day (hence deleted from the intersection), our protocol ensures that the deleted x only appears once in either Step 1b or Step 1c, but it should be hidden from

Initialization:

1. P_0 and P_1 independently generate key pairs for an additive homomorphic encryption scheme $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and $(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and share the public keys. Both parties agree on a randomly sampled PRF key $\mathsf{k} \xleftarrow{\$} \{0,1\}^\lambda$.
2. P_0 and P_1 generate initial trees with only an empty root and stash: $(\mathcal{D}_0, \mathcal{S}_0, \widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1)$ and $(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0, \mathcal{D}_1, \mathcal{S}_1)$, respectively. Initialize $I_0 = \emptyset$.

Day *d*: P_0 and P_1 hold $(\mathcal{D}_0, \mathcal{S}_0, \widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1)$ and $(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0, \mathcal{D}_1, \mathcal{S}_1)$, respectively. Let L_0 and L_1 be the heights of $\mathcal{D}_0$ (and $\widetilde{\mathcal{D}}_0$), and $\mathcal{D}_1$ (and $\widetilde{\mathcal{D}}_1$) respectively. Both parties update L_0 and L_1 as they update the trees below. Let X, Y denote the two parties' sets at the end of the previous day.
P_0 and P_1 have new input sets X_d^+, Y_d^+ which include elements they are adding to their set and X_d^-, Y_d^- of elements they are deleting. Denote $n^- = |X_d^-|$, $n^+ = |X_d^+|$, $m^- = |Y_d^-|$, $m^+ = |Y_d^+|$.

1. **Deletion:**
   (a) $\boldsymbol{X_d^-}$ **tree update.** P_0 sends $(\{(\widetilde{\mathsf{updates}}_i, \ell_i)\}_{i=1}^{n^-}, \widetilde{\mathcal{S}}_0') \leftarrow \mathsf{UpdateTree}(X_d^-, \{-x_i : x_i \in X_d^-\}_{i=1}^{n^-}, \mathcal{D}_0, \mathcal{S}_0, F_{\mathsf{k}}(\cdot), \mathsf{Enc}_{\mathsf{pk}_0}(\cdot))$ to P_1. P_1 replaces each path $\mathcal{P}(\ell_i)$ with $\widetilde{\mathsf{updates}}_i$ in $\widetilde{\mathcal{D}}_0$, and replaces $\widetilde{\mathcal{S}}_0$ with $\widetilde{\mathcal{S}}_0'$.
   (b) **Secret shares for** $\boldsymbol{X_d^- \cap Y}$**.** For all $x_i \in X_d^-$, run $\Pi_{\mathsf{CombinePath}}$ with P_0 as $\mathsf{Initiator}$ inputting $(x_i, -1, \widetilde{\mathsf{path}}_i \leftarrow \mathsf{GetPath}(\widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1, F_{\mathsf{k}}(\cdot), x_i))$ and P_1 as $\mathsf{Responder}$ inputting sk_1 corresponding to pk_1. They receive secret shares $[\![z_{x,i}^-]\!]_0$ and $[\![z_{x,i}^-]\!]_1$, respectively, where $z_{x,i}^- = -x_i$ if $x_i \in \mathcal{D}_1 \cup \mathcal{S}_1$ and 0 otherwise.
   (c) **Secret shares for** $\left(\boldsymbol{X \setminus X_d^-}\right) \cap \boldsymbol{Y_d^-}$**.** For all $y_j \in Y_d^-$, run $\Pi_{\mathsf{CombinePath}}$ with P_0 as $\mathsf{Responder}$ inputting sk_0 corresponding to pk_1 and P_1 as $\mathsf{Initiator}$ inputting $(y_j, -1, \widetilde{\mathsf{path}}_j \leftarrow \mathsf{GetPath}(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0, F_{\mathsf{k}}(\cdot), y_j))$. They receive secret shares $[\![z_{y,j}^-]\!]_0$ and $[\![z_{y,j}^-]\!]_1$, respectively, where $z_{y,j}^- = -y_j$ if $y_j \in \mathcal{D}_0 \cup \mathcal{S}_0$ and 0 otherwise.
   (d) $\boldsymbol{Y_d^-}$ **tree update.** P_1 sends $(\{(\widetilde{\mathsf{updates}}_j, \ell_j)\}_{j=1}^{m^-}, \widetilde{\mathcal{S}}_1') \leftarrow \mathsf{UpdateTree}(Y_d^-, \{-y_j : y_j \in Y_d^-\}_{j=1}^{m^-}, \mathcal{D}_1, \mathcal{S}_1, F_{\mathsf{k}}(\cdot), \mathsf{Enc}_{\mathsf{pk}_1}(\cdot))$ to P_0. P_0 replaces each path $\mathcal{P}(\ell_j)$ with $\widetilde{\mathsf{updates}}_j$ in $\widetilde{\mathcal{D}}_1$, and replaces $\widetilde{\mathcal{S}}_1$ with $\widetilde{\mathcal{S}}_1'$.
2. **Addition:**
   (a) $\boldsymbol{X_d^+}$ **tree update.** P_0 sends $(\{(\widetilde{\mathsf{updates}}_i, \ell_i)\}_{i=1}^{n^+}, \widetilde{\mathcal{S}}_0') \leftarrow \mathsf{UpdateTree}(X_d^+, \{x_i : x_i \in X_d^+\}_{i=1}^{n^+}, \mathcal{D}_0, \mathcal{S}_0, F_{\mathsf{k}}(\cdot), \mathsf{Enc}_{\mathsf{pk}_0}(\cdot))$ to P_1. P_1 replaces each path $\mathcal{P}(\ell_i)$ with $\widetilde{\mathsf{updates}}_i$ in $\widetilde{\mathcal{D}}_0$, and replaces $\widetilde{\mathcal{S}}_0$ with $\widetilde{\mathcal{S}}_0'$.
   (b) **Secret shares for** $\boldsymbol{X_d^+} \cap \left(\boldsymbol{Y \setminus Y_d^-}\right)$**.** For all $x_i \in X_d^+$, run $\Pi_{\mathsf{CombinePath}}$ with P_0 as $\mathsf{Initiator}$ inputting $(x_i, 1, \widetilde{\mathsf{path}}_i \leftarrow \mathsf{GetPath}(\widetilde{\mathcal{D}}_1, \widetilde{\mathcal{S}}_1, F_{\mathsf{k}}(\cdot), x_i))$ and P_1 as $\mathsf{Responder}$ inputting sk_1 corresponding to pk_1. They receive secret shares $[\![z_{x,i}^+]\!]_0$ and $[\![z_{x,i}^+]\!]_1$, respectively, where $z_{x,i}^+ = x_i$ if $x_i \in \mathcal{D}_1 \cup \mathcal{S}_1$ and 0 otherwise.
   (c) **Secret shares for** $\left(\boldsymbol{X \cup X_d^+ \setminus X_d^-}\right) \cap \boldsymbol{Y_d^+}$**.** For all $y_j \in Y_d^+$, run $\Pi_{\mathsf{CombinePath}}$ with P_0 as $\mathsf{Responder}$ inputting sk_0 corresponding to pk_0 and P_1 as $\mathsf{Initiator}$ inputting $(y_j, 1, \widetilde{\mathsf{path}}_j \leftarrow \mathsf{GetPath}(\widetilde{\mathcal{D}}_0, \widetilde{\mathcal{S}}_0, F_{\mathsf{k}}(\cdot), y_j))$. They receive secret shares $[\![z_{y,j}^+]\!]_0$ and $[\![z_{y,j}^+]\!]_1$, respectively, where $z_{y,j}^+ = y_j$ if $y_j \in \mathcal{D}_0 \cup \mathcal{S}_0$ and 0 otherwise.
   (d) $\boldsymbol{Y_d^+}$ **tree update.** P_1 sends $(\{(\widetilde{\mathsf{updates}}_j, \ell_j)\}_{j=1}^{m^-}, \widetilde{\mathcal{S}}_1') \leftarrow \mathsf{UpdateTree}(Y_d^+, \{y_j : y_j \in Y_d^+\}_{j=1}^{m^+}, \mathcal{D}_1, \mathcal{S}_1, F_{\mathsf{k}}(\cdot), \mathsf{Enc}_{\mathsf{pk}_1}(\cdot))$ to P_0. P_0 replaces each path $\mathcal{P}(\ell_j)$ with $\widetilde{\mathsf{updates}}_j$ in $\widetilde{\mathcal{D}}_1$, and replaces $\widetilde{\mathcal{S}}_1$ with $\widetilde{\mathcal{S}}_1'$.
3. **Output Generation:**
   (a) Let $\{[\![z_i]\!]_0\}_{i=1}^{\Gamma}$ and $\{[\![z_i]\!]_1\}_{i=1}^{\Gamma}$ be the shares received by P_0 and P_1 above, where $\Gamma = n^- + m^- + n^+ + m^+$. P_0 sends $\{\mathsf{Enc}_{\mathsf{pk}_0}([\![z_i]\!]_0)\}_{i=1}^{\Gamma}$ to P_1.
   (b) P_1 samples a random permutation π over $[\Gamma]$. P_1 samples a random mask $\alpha_i \xleftarrow{\$} \mathbb{Z}_q$ for each $i \in [\Gamma]$ and homomorphically adds them to the encryptions received from P_0. P_1 sends the following to P_0: $\pi\left(\{(\mathsf{Enc}_{\mathsf{pk}_0}([\![z_i]\!]_0) \oplus \mathsf{Enc}_{\mathsf{pk}_0}(\alpha_i)), [\![z_i]\!]_1 - \alpha_i)\}_{i=1}^{\Gamma}\right)$.
   (c) P_0 decrypts the first element in each pair using sk_0, and adds up each pair of shares to learn the shuffled set $\{z_j\}_{j=1}^{\Gamma}$.
   Output $I_d := I_{d-1} \cup \{z_j | z_j > 0\} \setminus \{-z_j | z_j < 0\}$.

Fig. 10. Protocol $\Pi_{\mathsf{UPSI\text{-}Del_{psi}}}$ for one-sided UPSI with addition and deletion functionality $\mathcal{F}_{\mathsf{UPSI\text{-}Del_{psi}}}$.

the parties whether the other party also deleted x on that day. To achieve this, the parties re-randomize and shuffle the results in Step 3.

4.3 Complexity, Correctness and Security

UPSI-Cardinalty/Sum with Addition and Deletion. Our protocols for $\Pi_{\mathsf{UPSI\text{-}Del}_{\mathsf{ca}}}$ and $\Pi_{\mathsf{UPSI\text{-}Del}_{\mathsf{sum}}}$ are presented in Fig. 9. On each day d, let N, M be the total number of additions and deletions of the two parties, respectively. Let the update set sizes be n and m, respectively. Then both the computation and communication complexity are $O(n \cdot (\sigma \cdot \log M + \rho) + m \cdot (\sigma \cdot \log N + \rho))$. We state the theorem below and defer its proof to the full version of our paper [15].

Theorem 2. *Assuming Π is a secure additively homomorphic encryption scheme, F is a pseudorandom function, the protocols $\Pi_{\mathsf{UPSI\text{-}Del}_{\mathsf{ca}}}, \Pi_{\mathsf{UPSI\text{-}Del}_{\mathsf{sum}}}$ presented in Fig. 9 securely realize the ideal functionalities $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{ca}}}, \mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{sum}}}$ defined in Fig. 6, respectively, against semi-honest adversaries.*

Plain UPSI with Addition and Deletion. We present our protocol $\Pi_{\mathsf{UPSI\text{-}Del}_{\mathsf{psi}}}$ in Fig. 10. On each day d, let N, M be the total number of additions and deletions of the two parties, respectively. Let the update set sizes be n and m, respectively. Then both the computation and communication complexity are $O(n \cdot (\sigma \cdot \log M + \rho) + m \cdot (\sigma \cdot \log N + \rho))$. We state the theorem below and defer its proof to the full version of our paper [15].

Theorem 3. *Assuming Π is a secure additively homomorphic encryption scheme, F is a pseudorandom function, the protocol $\Pi_{\mathsf{UPSI\text{-}Del}_{\mathsf{psi}}}$ presented in Fig. 10 securely realizes the ideal functionalities $\mathcal{F}_{\mathsf{UPSI\text{-}Del}_{\mathsf{psi}}}$ defined in Fig. 6 against semi-honest adversaries.*

5 Implementation Details and Optimizations

In this section, we discuss instantiations of the building blocks in our UPSI protocols and optimizations to further improve the concrete efficiency.

Encryption Schemes. In the addition-only UPSI protocols $\Pi_{\mathsf{UPSI\text{-}Add}_{\mathsf{ca}}}$ and $\Pi_{\mathsf{UPSI\text{-}Add}_{\mathsf{sum}}}$, we instantiate the $(2, 2)$-threshold additively homomorphic encryption scheme with exponential El Gamal encryption [29] to take advantage of efficient elliptic curve operations. Recall that in this scheme, $\mathsf{Enc}(m) = (g^r, h^r \cdot g^m)$ where the public key consists of a group generator g and a random group element $h = g^s$ with a secret key s. In the $(2, 2)$-threshold scheme, sk_0 and sk_1 form an additive secret share of s. Decryption of exponential El Gamal requires computing the discrete logarithm of a group element g^m, which is possible for a bounded message space. In all our addition-only UPSI protocols presented in Fig. 5, decryption occurs in Step 6. Observe that P_0 does *not* have to fully decrypt the first element in each tuple of m_3; instead, it is sufficient to check whether the decrypted message is 0 or not. In particular, given a partially decrypted ciphertext $\hat{c} = (a, b)$, P_0 can determine if the encrypted message is 0 by checking if $b = a^{\mathsf{sk}_0}$, without performing discrete logarithm. In $\Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$, P_0 needs to fully decrypt m_4', where the underlying message can be bounded by the maximum sum of associated values.

In $\Pi_{\mathsf{UPSI\text{-}Add_{circuit}}}$, while exponential El Gamal can still be used for the first ciphertext in m_3, the (masked) payload messages are distributed uniformly over the entire plaintext space, hence the payload messages are encrypted using $(2, 2)$-threshold Paillier encryption [43] instead.

In our protocols with both addition and deletion presented in Sect. 4 ($\Pi_{\mathsf{UPSI\text{-}Del_{psi}}}$ in Fig. 10 and $\Pi_{\mathsf{UPSI\text{-}Del_{ca}}}, \Pi_{\mathsf{UPSI\text{-}Del_{sum}}}$ in Fig. 9), El Gamal cannot be utilized because all the ciphertexts are encrypting secret shares that are distributed across the message space. Instead, the additively homomorphic encryption scheme is instantiated with Paillier. This has an impact on the computation time, as can be seen in Sect. 6.

Paillier Modulus Switching. Using Paillier in the deletion protocols introduces an additional technical challenge. Recall that the plaintext space in Paillier encryption is $\mathbb{Z}_n$ for a public key n, which is different for P_0's and P_1's keys. During our deletion protocols, parties perform $\Pi_{\mathsf{CombinePath}}$ for both $\mathsf{pk}_0 = n_0$ (P_0's public key) and $\mathsf{pk}_1 = n_1$ (P_1's public key) to get secret shares in both $\mathbb{Z}_{n_0}$ and $\mathbb{Z}_{n_1}$. We discuss how to combine these secret shares over different moduli.

Let ℓ be the maximum bit length required to represent a set element or associated value. Recall that if set elements are of arbitrary length, we can apply a hash function on all the elements and perform PSI on the hash outputs. In our evaluation section, each party holds at most 2^{22} elements, hence there are at most 2^{23} total elements. If we model the hash function as a random oracle, to ensure collision probability lower than $2^{-\kappa}$ for statistically security parameter $\kappa = 40$, it is safe to bound $\ell = 85$. Let n be a Paillier public key and L be the bit length of n, which is typically 1536 or 2048.

Consider a value $r \in \mathbb{Z}_{2^\ell}$ being secret shared as $[\![r]\!]_0, [\![r]\!]_1 \in \mathbb{Z}_n$. We will convert this secret share into another secret share of r in $\mathbb{Z}_{2^\ell}$. First, the integer summation of $[\![r]\!]_0 + [\![r]\!]_1$ is either r or $r + n$, and the probability $\Pr[[\![r]\!]_0 + [\![r]\!]_1 = r] \leq \Pr[[\![r]\!]_0 \leq r] \leq 2^{\ell - L} \ll 2^{-\kappa}$. Therefore, with overwhelming probability $[\![r]\!]_0 + [\![r]\!]_1 = r + n$. Let $s_0 = [\![r]\!]_0$ and $s_1 = [\![r]\!]_1 - n$, then $s_0 + s_1 = r$, where $s_0 > 0$ and $s_1 < 0$ as integers. If we represent s_1 in two's complement format, then the lowest ℓ bits of $s_0 + s_1$ should be r and the higher order bits should all be 0. Therefore, we can take the ℓ lowest order bits of s_0 and s_1 (in two's complement format) to form a secret share of r in $\mathbb{Z}_{2^\ell}$. Given that the original secret shares $[\![r]\!]_0, [\![r]\!]_1 \in \mathbb{Z}_n$ are distributed randomly over $\mathbb{Z}_n$, the new shares are statistically close to a uniform distribution over $\mathbb{Z}_{2^\ell}$ because $\ell \ll L$.

Realizing $\mathcal{F}_{\mathsf{lookup}}$. While $\mathcal{F}_{\mathsf{lookup}}$ can be instantiated with a generic secure two-party computation (2PC) protocol [32, 58], we construct a protocol that achieves better concrete efficiency, leveraging oblivious transfer (OT) and the efficient OT extension [12, 36]. Let (a, m_0, m_1) and b be the inputs to $\mathcal{F}_{\mathsf{lookup}}$ where m_0 is output when $a = b$ and m_1 otherwise. Before comparison, both parties compute a hash function $H : \mathbb{Z}_q \to \{0,1\}^{\ell_{\mathsf{gc}}}$ on their inputs a and b. The parties then run a garbled-circuit based equality testing to compute a binary secret share $[\![c]\!] \in \{0,1\}$ of $H(a) \overset{?}{=} H(b)$. Then two parties run an OT protocol where Sender inputs two messages $(m_{1-[\![c]\!]_0}, m_{[\![c]\!]_0})$ and Receiver inputs a choice bit $[\![c]\!]_1$. If $a = b$, then $[\![c]\!]_0 \neq [\![c]\!]_1$, in which case Receiver will receive m_0, as desired in $\mathcal{F}_{\mathsf{lookup}}$; if $a \neq b$, then $[\![c]\!]_0 = [\![c]\!]_1$ with overwhelming probability (see analysis below), and the Responder will receive m_1.

In this approach, we need the guarantee that if $a \neq b$, then $H(a) \neq H(b)$ with overwhelming probability, hence ℓ_{gc} should be sufficiently large. On the other hand, the size of the equality testing circuit grows with ℓ_{gc}, so we want to choose the smallest ℓ_{gc} such that the probability of a failure (i.e., that $H(a) = H(b)$ for $a \neq b$) over the entire protocol is less than $2^{-\kappa}$. In all the benchmarks presented in Sect. 6, there are at most 2^{23} elements held by both parties, and each element is compared against at most 2^9 elements in $\Pi_{\mathsf{CombinePath}}$. Hence the total number of $\mathcal{F}_{\mathsf{lookup}}$ invocations is bounded by $2^{23} \cdot 2^9 = 2^{32}$. The overall failure probability is no greater than $2^{32} \cdot 2^{-\ell_{gc}}$, and we want to ensure statistical security, namely $2^{32} \cdot 2^{-\ell_{gc}} \leq 2^{-\kappa}$ for $\kappa = 40$. Therefore, we set $\ell_{gc} \approx 32 + 40 = 72$.

6 Evaluation

6.1 Experimental Setup

We implement all of our UPSI protocols in C++ and report their performance in this section. We use the crypto library as part of Google's open-sourced Private Join and Compute project [7] for El Gamal and Paillier encryptions, Google's `gRPC` [2] for networking, and `emp-tool` [57] for instantiations of garbled circuits and oblivious transfer (including OT extension). Benchmarks are run on a Google Cloud [1] `c2-standard-16` virtual machine with 64 GB of RAM. Each party is executed on a single thread and communicate over `localhost`. The Linux `tc` command is used to simulate the various network settings. We simulate the LAN connection with 0.2 ms RTT network latency and 1Gbps network bandwidth. For WAN connection, we set the RTT latency to be 80 ms and test on various network bandwidths including 200 Mbps, 50 Mbps, and 5 Mbps. Our implementation is available on GitHub: https://github.com/ruidazeng/upsi-revisited.

Addition-Only UPSI. To demonstrate the updatable property of our protocols, we consider the setting where both parties begin with an empty set to which N_d elements are added each day. Our benchmarks represent the performance of the protocols on day $(\frac{N}{N_d})$ where the size of each party's set reaches N.

We compare our plain UPSI protocols with the state-of-the-art semi-honest PSI protocol [51] (RR22), and compare our UPSI for extended functionalities (PSI-Cardinality, PSI-Sum, and Circuit-PSI) with the state-of-the-art Circuit-PSI [20] (CGS22) and [51] (RR22), where, on day $(\frac{N}{N_d})$, the parties run PSI or Circuit-PSI on their full input sets of size N. Note that the Circuit-PSI protocols [20,51] are also state-of-the-art for computing PSI-Cardinality or PSI-Sum, with slight modifications to their protocols. In our comparison, we assume these modifications do not incur extra overhead in their performance. We also compare our addition-only plain UPSI with [16] to demonstrate the improvement of worst-case complexity by plugging in our new oblivious data structure.

We don't compare with the protocols specifically designed for PSI-Cardinality or PSI-Sum [30,35] because these protocols are outperformed by [20,51]. A more recent work [18] improves PSI and Circuit-PSI communication by 12% compared to [51], but we don't compare with it for three reasons: (1) their construction is built on the Silver codes [26], which turns out to be insecure [52], (2) their source code is not available online, and (3) even if their construction is instantiated with secure codes, from our comparison with the other works, we expect our protocols to perform better in certain settings as well. Note that [51] is also instantiated with the insecure Silver codes, but their open-sourced library [50] supports instantiating the construction with the state-of-the-art OT extension from expand-accumulate codes [19], which is what we compare with.

UPSI with Addition and Deletion. In the setting with both addition and deletion, standalone PSI protocols need only compute over elements that remain in the input sets. In the extreme case where the every element is added and then soon deleted, the input sets remain small and so the standalone PSI protocols would likely be optimal. Alternatively, if the input sets are growing at a steady rate, then our constructions may be best. These caveats should be understood and application-specific context would play a role in choosing a solution.

In our benchmarks, we assume roughly 3/4 set operations are additions and 1/4 are deletions. We further assume that each element can only be added and deleted *at most once* in each party's set (i.e., an element cannot be re-added once it has been deleted). In this case, the computation and communication complexity of our protocols are $O(N_d \cdot \log N)$.

Choice of N and N_d. In all of our experiments, we chose the values for N and N_d that would best demonstrate the *turning point* where we become competitive. Our protocols have more advantages when increasing the gap between N and N_d. As N increases (e.g., for billion-sized sets [14,38]), we expect our protocol to be dominant for more network settings and larger N_d values. In all of our comparison tables, cells in green indicate the state-of-the-art performance, and those in blue indicate that our protocols perform better.

Concrete Parameters. We set the computational security parameter $\lambda = 128$ and the statistical security parameter $\kappa = 40$. Following the analysis in [55], we set the maximum tree node capacity $\sigma = 4$ and the maximum stash capacity $\rho = 89$ to achieve failure probability of 2^{-80} for inserting a single element into the tree. Even with our largest set size of 2^{22}, the combined failure probability is bounded well below $2^{-\kappa}$. In protocols with addition and deletion, we allow parties to add and delete each element at most once, and so we double both our node size (to $\sigma = 8$) and stash size (to $\rho = 178$), and we defer the analysis to the full version of our paper [15]. To enable P_0 to efficiently decrypt m'_4 in Step

6 of $\Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$ (Fig. 5) with exponential El Gamal encryption, we bound the PSI-Sum maximum value to be at most 10,000. Larger sums can either utilize extra storage with a lookup table or switch to using Paillier encryption.

6.2 Addition-Only UPSI with Extended Functionalities

We compare our addition-only UPSI for extended functions (PSI-Cardinality, PSI-Sum, and Circuit-PSI) with [51] (RR22) and [20] (CGS22) in Table 2 with total set sizes ranging from 2^{18} to 2^{22} and update sizes from 2^6 to 2^{10}. Our computation and communication complexity grows logarithmically with the total set size and linearly with the update size N_d, so our protocols are more competitive in larger input sizes and smaller update sizes. Note that [20] (CGS22) presents two constructions ($\mathsf{C\text{-}PSI}_1$ and $\mathsf{C\text{-}PSI}_2$) with different trade offs between computation and communication, but for all the parameters we choose, $\mathsf{C\text{-}PSI}_2$ outperforms $\mathsf{C\text{-}PSI}_1$ in all aspects. We were unable to run CGS22 with input size of 2^{22}, so we use the communication cost and running time under LAN reported in their paper [20], and estimate the running time in the WAN settings.

Communication: Since our communication grows linearly with N_d and only logarithmically with N, our protocols have a communication advantage in settings where $N_d \ll N$. For $N = 2^{18}$, our communication has an improvement of $2.2-13\times$ when $N_d = 2^6$ in all functionalities, and when $N_d = 2^8$, $\Pi_{\mathsf{UPSI\text{-}Add_{ca}}}$ and $\Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$ have an advantage $1.8 - 3.4\times$. For $N = 2^{20}$, our protocols outperform RR22 by $2.2 - 50\times$ depending on the functionality and update size, with only $\Pi_{\mathsf{UPSI\text{-}Add_{circuit}}}$ at $N_d = 2^{10}$ not showing an improvement. When $N = 2^{22}$, that improvement extends to all settings and increases to a factor of $2.2-200\times$.

Computation: Our computational complexity also grows linearly with N_d and logarithmically with N. Despite this, our computation times do not reflect this asymptotic improvement as clearly, which stems from our usage of costly public key operations. As a result, we show better performance only when N is sufficiently large. In the LAN setting with $N = 2^{20}, N_d = 2^6$, our $\Pi_{\mathsf{UPSI\text{-}Add_{ca}}}$ and $\Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$ are faster by $3.2\times$ and $2.1\times$, respectively. By $N = 2^{22}, N_d = 2^6 - 2^8$, our $\Pi_{\mathsf{UPSI\text{-}Add_{ca}}}, \Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$ protocols outperform CGS22 by $1.4 - 15\times$.

End to End: Given these communication and computation trade offs, our protocols perform best with more realistic network configurations with lower network bandwidth. At $N = 2^{18}$, we begin to have competitive runtimes for $\Pi_{\mathsf{UPSI\text{-}Add_{ca}}}$ and $\Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$ in the smaller update size $N_d = 2^6$. By $N = 2^{22}$ and $N_d = 2^6$, our protocols outperform in all network settings by $15 - 76\times$ for $\Pi_{\mathsf{UPSI\text{-}Add_{ca}}}$, $11 - 46\times$ for $\Pi_{\mathsf{UPSI\text{-}Add_{sum}}}$, and $1.8 - 9.4\times$ for $\Pi_{\mathsf{UPSI\text{-}Add_{circuit}}}$.

6.3 UPSI-Cardinality/Sum with Addition and Deletion

Our performance for $\Pi_{\mathsf{UPSI\text{-}Del_{ca}}}$ and $\Pi_{\mathsf{UPSI\text{-}Del_{sum}}}$ in comparison with [20,51] is presented in Table 3. Since the two protocols are implemented in the same way except that P_0's inputting payloads are different, they have close experimental

Table 2. Communication cost (in MB) and running time (in seconds) comparing our addition-only UPSI protocols to prior work. * indicates estimated communication and running time.

N	N_d	Protocol	Comm. (MB)	Total Running Time (s)			
				LAN	200Mbps	50Mbps	5Mbps
2^{18}	–	RR22	37.1	7.76	10.7	13.8	64.4
		CGS22 (C-PSI$_1$)	548	7.90	36.9	106	968
		CGS22 (C-PSI$_2$)	353	6.32	29.4	70.6	619
	2^6	$\Pi_{\text{UPSI-Add}_{\text{ca}}}$	2.83	7.12	7.59	7.87	11.8
	2^8		11.0	27.6	28.6	30.2	45.6
	2^{10}		42.6	108	110	115	177
	2^6	$\Pi_{\text{UPSI-Add}_{\text{sum}}}$	5.35	11.0	11.8	12.5	20.1
	2^8		22.3	45.9	47.2	49.3	82.0
	2^{10}		87.1	178	184	195	321
	2^6	$\Pi_{\text{UPSI-Add}_{\text{circuit}}}$	17.1	81.7	83.1	85.3	110
	2^8		67.0	318	327	330	427
	2^{10}		248	1171	1182	1214	1570
2^{20}	–	RR22	149	31.1	38.4	51.9	258
		CGS22 (C-PSI$_1$)	2190	31.0	135	414	3771
		CGS22 (C-PSI$_2$)	1408	24.3	92.8	268	3872
	2^6	$\Pi_{\text{UPSI-Add}_{\text{ca}}}$	3.03	7.59	8.14	8.46	12.6
	2^8		11.8	29.6	30.6	32.0	48.7
	2^{10}		45.7	116	121	127	194
	2^6	$\Pi_{\text{UPSI-Add}_{\text{sum}}}$	5.70	11.8	12.5	13.1	21.5
	2^8		22.3	45.9	47.2	49.3	82.0
	2^{10}		87.1	178	184	195	321
	2^6	$\Pi_{\text{UPSI-Add}_{\text{circuit}}}$	17.1	81.7	83.1	85.3	110
	2^8		67.0	318	327	330	427
	2^{10}		264	1251	1263	1295	1674
2^{22}	–	RR22	606	125	159	214	1086
		CGS22 (C-PSI$_1$)	6667*	93.0*	126*	226*	1426*
		CGS22 (C-PSI$_2$)	4435*	77.9*	100*	167*	965*
	2^6	$\Pi_{\text{UPSI-Add}_{\text{ca}}}$	3.22	8.09	9.02	9.33	14.3
	2^8		12.6	31.6	32.7	34.2	52.7
	2^{10}		48.9	123	127	133	205
	2^6	$\Pi_{\text{UPSI-Add}_{\text{sum}}}$	6.04	12.5	13.3	14.1	23.6
	2^8		23.6	48.8	50.3	53.3	88.6
	2^{10}		92.7	191	197	209	342
	2^6	$\Pi_{\text{UPSI-Add}_{\text{circuit}}}$	18.1	86.6	88.4	90.2	116
	2^8		71.1	339	343	352	454
	2^{10}		280	1348	1341	1376	1780

results. We combine them in the table. This protocol is more expensive than the addition-only ones, so we set smaller update sizes of $N_d = 2^4, 2^5, 2^6$ to demonstrate the turning point where our protocols start to perform better. Our experiments for input size $N = 2^{22}$ are run on a Google Cloud `c2-standard-30` virtual machine with 120 GB of RAM as we run out of 64 GB memory.

Table 3. Communication cost (in MB) and running time (in seconds) of our protocols for UPSI-Cardinality and UPSI-Sum with addition and deletion in comparison with prior work. * indicates estimated communication and running time.

N	N_d	Protocol	Comm. (MB)	Total Running Time (s)			
				LAN	200Mbps	50Mbps	5Mbps
2^{20}	–	RR22	149	31.1	38.4	51.9	258
		CGS22 (C-PSI$_1$)	2190	31.0	135	414	3771
		CGS22 (C-PSI$_2$)	1408	24.3	92.8	415	3872
	2^4	$\Pi_{\mathsf{UPSI\text{-}Del_{ca}}}$ $\Pi_{\mathsf{UPSI\text{-}Del_{sum}}}$	58.5	96.1	101	106	179
	2^5		116	190	198	212	362
	2^6		231	364	375	402	723
2^{22}	–	RR22	606	125	159	214	1086
		CGS22 (C-PSI$_1$)	6667*	93.0*	126*	226*	1426*
		CGS22 (C-PSI$_2$)	4435*	77.9*	100*	167*	965*
	2^4	$\Pi_{\mathsf{UPSI\text{-}Del_{ca}}}$ $\Pi_{\mathsf{UPSI\text{-}Del_{sum}}}$	61.4	103	107	113	191
	2^5		122	203	210	223	383
	2^6		243	385	399	429	765

Communication: Our communication complexity is $O(N_d \cdot \log N)$, but the improvements are not as stark, for two reasons: (1) the increased stash and node sizes required, and (2) in addition to exchanging ciphertexts, the parties also perform OT and garbled circuits. Despite this, our protocol still achieves lower communication overhead in most settings. At $N = 2^{20}$, our communication has an improvement of $1.3 - 2.5\times$ when $N_d \leq 2^5$. By $N = 2^{22}$, our communication has an improvement of $2.5 - 9.9\times$ for all update sizes.

Computation: Our performance under LAN is again dominated by public key operations, but, unlike in the addition-only protocols, does not benefit from the efficient El Gamal instantiations. Our computation has the same growth rate as communication, and so we expect our performance to eventually beat CGS22 when N is sufficiently large.

End to End: As shown in Table 3, the end to end running time of our protocol begins to outperform RR22 and CGS22 at 5 Mbps when $N = 2^{20}, N_d = 2^4$ by

$1.4\times$. By $N = 2^{22}$, we show an improvement of $1.3 - 5.1\times$ at 5 Mbps for all update sizes, and an improvement of $1.5\times$ at 50 Mbps for $N_d = 2^4$.

6.4 UPSI for Plain PSI

We compare our plain UPSI protocols with [51] (RR22) in Table 4. We evaluate two constructions in RR22 with different encoding sizes of $1.28n$ and $1.23n$, which have different trade offs in computation and communication, denoted as `fast` and `small` respectively in the table. Note that our addition-only plain UPSI (Fig. 5) contains only one encrypted tree, hence it is more efficient than our other addition-only protocols. To best demonstrate our turning point, we use $N_d = 2^4, 2^6, 2^8, 2^{10}$ for $\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$ and $N_d = 2^4, 2^5, 2^6$ for $\Pi_{\mathsf{UPSI\text{-}Del_{psi}}}$

Table 4. Communication cost (in MB) and running time (in seconds) of our protocols for plain UPSI in comparison with prior work.

N	N_d	Protocol	Comm. (MB)	Total Running Time (s)			
				LAN	200Mbps	50Mbps	5Mbps
2^{20}	–	RR22 (fast)	34.3	0.73	3.09	7.10	55.9
		RR22 (small)	32.1	1.00	3.21	6.97	52.8
	2^4	$\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$	0.50	1.41	1.84	1.89	2.48
	2^6		1.95	5.54	6.11	6.30	8.88
	2^8		7.57	21.6	22.8	23.5	34.1
	2^{10}		29.6	84.9	87.5	90.8	133
	2^4	$\Pi_{\mathsf{UPSI\text{-}Del_{psi}}}$	58.7	98.6	103	109	181
	2^5		117	195	203	215	369
	2^6		231	370	384	410	729
2^{22}	–	RR22 (fast)	138	3.45	11.3	27.7	227
		RR2 (small)	129	4.81	12.2	27.6	214
	2^4	$\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$	0.53	1.49	1.93	1.97	2.57
	2^6		2.06	5.89	6.48	6.67	9.51
	2^8		8.03	22.9	24.1	24.9	36.2
	2^{10}		31.5	89.9	92.8	96.2	141
	2^4	$\Pi_{\mathsf{UPSI\text{-}Del_{psi}}}$	61.6	105	109	115	194
	2^5		122	208	214	228	388
	2^6		243	396	412	437	776

Communication: Similarly as in our other protocols, our communication complexity in both $\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$ and $\Pi_{\mathsf{UPSI\text{-}Del_{psi}}}$ are $O(N_d \cdot \log N)$. The communication cost of $\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$ outperforms RR22 by $1.1 - 240\times$ in all settings, whereas that of $\Pi_{\mathsf{UPSI\text{-}Del_{psi}}}$ only beats RR22 by $1.1 - 2.1\times$ with $N = 2^{22}$ and $N_d = 2^4, 2^5$.

Computation: Our computation complexity is similar to communication, leading to better performance when N is sufficiently large. Our addition-only protocol starts to outperforms RR22 when $N = 2^{22}$ and $N_d = 2^4$.

End to End: As the communication and computation discussed above, our protocols are more competitive with larger input sizes, smaller updates, and in networks with lower bandwidths. By $N = 2^{22}$ and $N_d = 2^4$, $\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$ achieves an improvement of $2.3 - 88\times$ in all network settings. It outperforms RR22 by $1.5\times$ even when the update size grows to 2^{10}.

6.5 Worst-Case Logarithmic Complexity

We compare our one-sided addition-only plain UPSI protocol $\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$ with that of [16] (BMX22). While BMX22 has amortized complexity of $O(N_d \cdot \log N)$, their worst-case complexity is $O(N)$ when they update the leaf level of the tree. By plugging in our new oblivious data structure, we significantly reduce the worst-case complexity to $O(N_d \cdot \log N)$. The worst-case performance (Max) and amortized performance (Avg) are presented in Table 5 with $N = 2^{18}, 2^{20}$ and $N_d = 2^6, 2^8, 2^{10}$. To analyze the amortized cost of BMX22, we start with two sets each of size N. Then, on every new day, both parties add a new set of size

Table 5. Communication cost (in MB) and running time (in seconds) comparing our addition-only plain UPSI protocol to the worst-case and average-case performance of [16].

N	N_d	Protocol	Comm.(MB)		Total Running Time(s)							
			Max	Avg	LAN		200Mbps		50Mbps		5Mbps	
					Max	Avg	Max	Avg	Max	Avg	Max	Avg
2^{18}	2^6	BMX22	120	1.09	79.6	4.30	85.9	4.53	100	4.59	272	5.88
		$\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$	1.82		5.17		6.24		6.31		8.70	
	2^8	BMX22	121	3.74	77.9	14.7	84.2	15.1	98.3	15.5	268	20.3
		$\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$	7.08		20.2		21.8		22.6		32.4	
	2^{10}	BMX22	122	12.5	86.4	49.0	87.7	49.9	95.1	51.3	268	67.2
		$\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$	27.7		79.4		81.5		84.7		124	
2^{20}	2^6	BMX22	480	1.25	321	4.92	350	5.17	403	5.24	1090	6.76
		$\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$	1.95		5.54		6.11		6.30		8.88	
	2^8	BMX22	481	4.37	319	17.2	344	17.6	401	18.1	1090	23.7
		$\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$	7.57		21.6		22.8		23.5		34.1	
	2^{10}	BMX22	482	15.0	312	58.9	337	59.9	394	61.4	1090	81.1
		$\Pi_{\mathsf{UPSI\text{-}Add_{psi}}}$	29.6		84.9		87.5		90.8		133	

N_d to their existing sets and run the UPSI protocol. We repeat this process over a period of several days ($\frac{N}{N_d}$) until the total set size of each party reaches $2N$. We report the amortized cost over these $\frac{N}{N_d}$ days.

Comparison. As shown in Table 5, our communication cost is comparable to BMX22's average-case while outperforming their worst-case by $4.4 - 246\times$ in all settings since their worst-case communication grows linearly with N. Similarly, our computation cost is comparable to their average-case while outperforming their worst-case by $1.1 - 58\times$ in the LAN setting. As a result, the end to end running time of our protocol outperforms BMX22's worst-case in all settings by $1.1 - 123\times$, while having $1.1 - 1.8\times$ overhead compared to their average-case. Concerning the worst-case performance, our protocol has more advantages in larger input sizes and smaller updates.

Acknowledgments. This project is supported in part by the NSF CNS Award 2247352, Brown Data Science Seed Grant, Meta Research Award, Google Research Scholar Award, and Amazon Research Award.

References

1. Google Cloud. https://cloud.google.com.
2. Google Remote Procedure Call (gPRC). https://grpc.io.
3. Password Monitor: Safeguarding passwords in Microsoft Edge. https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/.
4. Password Monitoring – Apple Platform Security. https://support.apple.com/en-al/guide/security/sec78e79fc3b/web.
5. Privacy-Preserving Contact Tracing. https://covid19.apple.com/contacttracing.
6. Private Intersection-Sum Protocols with Applications to Attributing Aggregate Ad Conversions. https://research.google/pubs/pub51026/.
7. Private Join and Compute. https://github.com/google/private-join-and-compute.
8. Protect your accounts from data breaches with Password Checkup. https://security.googleblog.com/2019/02/protect-your-accounts-from-data.html.
9. Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/.
10. Aydin Abadi, Changyu Dong, Steven J. Murdoch, and Sotirios Terzis. Multi-party updatable delegated private set intersection. In Ittay Eyal and Juan A. Garay, editors, *FC 2022*, volume 13411 of *LNCS*, pages 100–119. Springer, Cham, May 2022.
11. Archita Agarwal, David Cash, Marilyn George, Seny Kamara, Tarik Moataz, and Jaspal Singh. Updatable private set intersection from structured encryption. Cryptology ePrint Archive, 2024. https://eprint.iacr.org/2024/1183.
12. Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 535–548. ACM Press, November 2013.
13. Giuseppe Ateniese, Emiliano De Cristofaro, and Gene Tsudik. (If) size matters: Size-hiding private set intersection. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *PKC 2011*, volume 6571 of *LNCS*, pages 156–173. Springer, Berlin, Heidelberg, March 2011.

14. Saikrishna Badrinarayanan, Ranjit Kumaresan, Mihai Christodorescu, Vinjith Nagaraja, Karan Patel, Srinivasan Raghuraman, Peter Rindal, Wei Sun, and Minghua Xu. A plug-n-play framework for scaling private set intersection to billion-sized sets. In *Cryptology and Network Security - 22nd International Conference, CANS 2023, Augusta, GA, USA, October 31 - November 2, 2023, Proceedings*, volume 14342 of *Lecture Notes in Computer Science*, pages 443–467. Springer, 2023.
15. Saikrishna Badrinarayanan, Peihan Miao, Xinyi Shi, Max Tromanhauser, and Ruida Zeng. Updatable private set intersection revisited: Extended functionalities, deletion, and worst-case complexity. Cryptology ePrint Archive, 2024. https://eprint.iacr.org/2024/1446.
16. Saikrishna Badrinarayanan, Peihan Miao, and Tiancheng Xie. Updatable private set intersection. *PoPETs*, 2022(2):378–406, April 2022.
17. Alex Berke, Michiel A. Bakker, Praneeth Vepakomma, Ramesh Raskar, Kent Larson, and Alex 'Sandy' Pentland. Assessing disease exposure risk with location histories and protecting privacy: A cryptographic approach in response to A global pandemic. *CoRR*, abs/2003.14412, 2020.
18. Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. Near-optimal oblivious key-value stores for efficient psi, PSU and volume-hiding multi-maps. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association, 2023.
19. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 603–633. Springer, Cham, August 2022.
20. Nishanth Chandran, Divya Gupta, and Akash Shah. Circuit-PSI with linear complexity via relaxed batch OPPRF. *PoPETs*, 2022(1):353–372, January 2022.
21. Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 34–63. Springer, Cham, August 2020.
22. Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1223–1237. ACM Press, October 2018.
23. Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1243–1255. ACM Press, October / November 2017.
24. Wutichai Chongchitmate, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. PSI from ring-OLE. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 531–545. ACM Press, November 2022.
25. Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1135–1150. ACM Press, November 2021.
26. Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 502–534, Virtual Event, August 2021. Springer, Cham.

27. Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 143–159. Springer, Berlin, Heidelberg, January 2010.
28. Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 789–800. ACM Press, November 2013.
29. Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
30. Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. Private set operations from oblivious switching. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 591–617. Springer, Cham, May 2021.
31. Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 395–425, Virtual Event, August 2021. Springer, Cham.
32. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.
33. Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.
34. Bernardo A. Huberman, Matthew K. Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In Stuart I. Feldman and Michael P. Wellman, editors, *Proceedings of the First ACM Conference on Electronic Commerce (EC-99), Denver, CO, USA, November 3-5, 1999*, pages 78–86. ACM, 1999.
35. Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. On deploying secure computing: Private intersection-sum-with-cardinality. In *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*, pages 370–389. IEEE, 2020.
36. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Berlin, Heidelberg, August 2003.
37. Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 1447–1464. USENIX Association, August 2019.
38. Seny Kamara, Payman Mohassel, Mariana Raykova, and Saeed Sadeghian. Scaling private set intersection to billion-element sets. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *Financial Cryptography and Data Security*, pages 195–215, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
39. Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 818–829. ACM Press, October 2016.
40. Catherine Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 7-9, 1986*, pages 134–137. IEEE Computer Society, 1986.

41. Peihan Miao, Sarvar Patel, Mariana Raykova, Karn Seth, and Moti Yung. Two-sided malicious security for private intersection-sum with cardinality. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 3–33. Springer, Cham, August 2020.
42. Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-N OT extension with application to private set intersection. In Helena Handschuh, editor, *CT-RSA 2017*, volume 10159 of *LNCS*, pages 381–396. Springer, Cham, February 2017.
43. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.
44. Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-N OT extension with application to private set intersection. In Helena Handschuh, editor, *CT-RSA 2017*, volume 10159 of *LNCS*, pages 381–396. Springer, Cham, February 2017.
45. Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from PaXoS: Fast, malicious private set intersection. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 739–767. Springer, Cham, May 2020.
46. Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 515–530. USENIX Association, August 2015.
47. Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 122–153. Springer, Cham, May 2019.
48. Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 125–157. Springer, Cham, April / May 2018.
49. Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 797–812. USENIX Association, August 2014.
50. Srinivasan Raghuraman and Peter Rindal. VOLE-PSI. https://github.com/Visa-Research/volepsi.
51. Srinivasan Raghuraman and Peter Rindal. Blazing fast PSI from improved OKVS and subfield VOLE. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2505–2517. ACM Press, November 2022.
52. Srinivasan Raghuraman, Peter Rindal, and Titouan Tanguy. Expand-convolute codes for pseudorandom correlation generators from LPN. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 602–632. Springer, Cham, August 2023.
53. Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 235–259. Springer, Cham, April / May 2017.

54. Peter Rindal and Phillipp Schoppmann. VOLE-PSI: Fast OPRF and circuit-PSI from vector-OLE. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 901–930. Springer, Cham, October 2021.
55. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.
56. Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. *IEEE Data Eng. Bull.*, 43(2):95–107, 2020.
57. Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.
58. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

Honest Majority GOD MPC with $O(\mathsf{depth}(C))$ Rounds and Low Online Communication

Amit Agarwal[1(✉)], Alexander Bienstock[2], Ivan Damgård[3], and Daniel Escudero[2]

[1] University of Illinois Urbana-Champaign, Champaign, USA
amita2@illinois.edu
[2] J.P. Morgan AI Research and J.P. Morgan AlgoCRYPT CoE, New York, USA
[3] Aarhus University, Aarhus, Denmark

Abstract. In the context of secure multiparty computation (MPC) protocols with guaranteed output delivery (GOD) for the honest majority setting, the state-of-the-art in terms of communication is the work of (Goyal *et al.* CRYPTO'20), which communicates $O(n|C|)$ field elements, where $|C|$ is the size of the circuit being computed and n is the number of parties. Their round complexity, as usual in secret-sharing based MPC, is proportional to $O(\mathsf{depth}(C))$, but only in the optimistic case where there is no cheating. Under attack, the number of rounds can increase to $\Omega(n^2)$ before honest parties receive output, which is undesired for shallow circuits with $\mathsf{depth}(C) \ll n^2$. In contrast, other protocols that only require $O(\mathsf{depth}(C))$ rounds *even in the worst case* exist, but the state-of-the-art from (Choudhury and Patra, Transactions on Information Theory, 2017) still requires $\Omega(n^4|C|)$ communication in the offline phase, and $\Omega(n^3|C|)$ in the online (for both point-to-point and broadcast channels). We see there exists a tension between efficient communication and number of rounds. For reference, the recent work of (Abraham et al., EUROCRYPT'23) shows that for perfect security and $t < n/3$, protocols with both linear communication and $O(\mathsf{depth}(C))$ rounds exist.

We address this state of affairs by presenting a novel honest majority GOD protocol that maintains $O(\mathsf{depth}(C))$ rounds, *even under attack*, while improving over the communication of the most efficient protocol in this setting by Choudhury and Patra. More precisely, our protocol has point-to-point (P2P) online communication of $O(n|C|)$, accompanied by $O(n|C|)$ broadcasted (BC) elements, while the offline has $O(n^3|C|)$ P2P communication with $O(n^3|C|)$ BC. This improves over the previous best result, and reduces the tension between communication and round complexity. Our protocol is achieved via a careful use of packed secret-sharing in order to improve the communication of existing verifiable secret-sharing approaches, although at the expense of weakening their robust guarantees: reconstruction of shared values may fail, but only if the adversary gives away the identities of many corrupt parties. We show that this less powerful notion is still useful for MPC, and we use this as a core building block in our construction. Using this weaker

© International Association for Cryptologic Research 2025
K.-M. Chung and Y. Sasaki (Eds.): ASIACRYPT 2024, LNCS 15489, pp. 234–265, 2025.
https://doi.org/10.1007/978-981-96-0938-3_8

VSS, we adapt the recent secure-with-abort Turbopack protocol (Escudero *et al.* CCS'22) to the GOD setting without significantly sacrificing in efficiency.

1 Introduction

Secure multiparty computation, or MPC for short, is a set of tools that enable a set of parties $P_1, \ldots, P_n$, each P_i holding an input x_i, to compute any function $y = f(x_1, \ldots, x_n)$ while revealing only the output y. This holds even if an adversary corrupts t out of the n parties, and if the protocol is what is known as *actively secure*, then security holds even if the corrupt parties deviate arbitrarily from the protocol execution. A particularly interesting setting, which is the focus of this work, is the *honest majority* case where it is assumed that $t < n/2$, that is, the adversary only corrupts a minority of the parties. In this case, the strong property of *guaranteed output delivery* (G.O.D.) can be achieved, which ensures the honest parties receive the output y in spite of any malicious behavior by the corrupt parties. Furthermore, if one is willing to assume the existence of a broadcast channel, the whole protocol can be made unconditionally secure, meaning it is resistant against unbounded adversaries and it only has a negligible amount of error. This is in contrast to *security with abort*, which ensures privacy but may not guarantee output provision.

G.O.D. is the strongest security notion for MPC and very important in practice as it removes all problems with deciding who failed if the protocol aborts. Hence, optimizing the efficiency of G.O.D. protocols has been a topic of study in recent literature. First, as usual in MPC, let us model the computation to be carried out as an arithmetic circuit C over a large-enough finite field $\mathbb{F}$. Let us begin by discussing the case in which $t < n/3$, which is *weaker* than honest majority $t < n/2$ as it tolerates less corruptions. Earlier works such as BGW [7] and (concurrently) [9] show that G.O.D. is possible for $t < n/3$ with perfect security, having a communication complexity (measured in the total number of field elements) of $O(n^4|C|)$. Several follow-up works were devoted to improving the communication. Towards such goal, [19] introduces the *player elimination* framework, which reduces communication to $O(n^3|C|)$, and building on top of this framework, the works of [6,17] get linear communication $O(n|C|)$. Unfortunately, the number of rounds—which is also an important metric directly related to end-to-end performance—of protocols based on player elimination is proportional to $O(\mathsf{depth}(C))$, but only in the *optimistic case* where there is no cheating. When the corrupt parties misbehave, the protocol enters a "recovery state" and eventually resumes after a previous "checkpoint". This process can happen $\Omega(n)$ times, and overall it adds $\Omega(n)$ rounds to the round-count, which is particularly impactful when $\mathsf{depth}(|C|) = o(n)$. For high latency settings such as wide area networks this can drastically affect the performance of the protocol. Studying the communication of $t < n/3$ MPC with $O(\mathsf{depth}(C))$ rounds (independently of n, even in the worst case) has been a topic of study of recent works: [1] achieved $O(n^3|C|)$ communication, and the recent work of [2] lowered it to $O(n|C|)$, matching (asymptotically) that of the previous state-of-the-art [6,17], but crucially, without paying any additive overhead in n on the number of rounds.

The above discussion is for the $t < n/3$ setting, which is considerably easier than the $n/3 < t < n/2$ regime. For $t < n/2$, it is known that G.O.D. is possible *assuming a broadcast channel*, and the early work by Rabin and Ben-Or [24] achieved $O(\mathsf{depth}(C))$ rounds even in the worst case, but it is very expensive in terms of communication. For instance, its communication is higher than that of [11], which is $O(n^5|C|) + O(n^3) \times \mathsf{BC}$, where the term multiplying the BC denotes the number of elements sent over the broadcast channel. The work of [11] however has a number of rounds that, in the worst case, can become $O(n \cdot \mathsf{depth}(C))$. In the last two decades several subsequent works approached the task of improving communication [5,8,18], culminating in the current state-of-the-art by Goyal, Song, and Zhu [18], which has $O(n|C| + O(n^3) \times \mathsf{BC})$ communication. All these works require $O(\mathsf{depth}(|C|))$ rounds in the optimistic case, but in case of cheating, the "recovery state" adds not only $\Omega(n)$, but $\Omega(n^2)$ extra rounds,[1] which is extremely harmful for circuits with $\mathsf{depth}(C) \ll n^2$. Currently, and unlike the $t < n/3$, removing this round-count overhead can only be done at the expense of increasing communication. Indeed, the most recent work exploring the task of $O(\mathsf{depth}(C))$-round honest majority G.O.D. MPC is [10], which has a communication of $O(n^4|C|) + O(n^4|C|) \times \mathsf{BC}$ in total, with an online phase of $O(n^3|C|) + O(n^3|C|) \times \mathsf{BC}$.

Towards improving this state of affairs, recent work by Escudero and Fehr [14] presents an honest majority G.O.D. protocol that achieves $O(\mathsf{depth}(C))$ rounds, independent of n, and simultaneously has linear communication $O(n|C|) + 0 \times \mathsf{BC}$. However, this is done in the *preprocessing model* where the parties are assumed to have certain correlated randomness independent of their inputs available for free. When instantiating this preprocessing with a protocol that has number of rounds independent of n—like the one from [10]—the resulting preprocessing complexity would be $O(|C|n^6) + O(|C|n^6) \times \mathsf{BC}$, which is too large. This situation leads to the following interesting and challenging question:

How communication-efficient can honest majority G.O.D. protocols be, while using only $O(\mathsf{depth}(|C|))$ rounds even under attack (that is, independent of n)?

1.1 Our Contribution

In this work we make progress on this question by providing an MPC protocol in the honest majority case $t < n/2$ that has $O(\mathsf{depth}(C))$ rounds even in the worst case, while improving over the communication of the state-of-the-art [10] in this regime. We achieve a communication of $O(n|C| + \mathsf{depth}(C)n^3) + O(n|C| + \mathsf{depth}(C)n^3) \times \mathsf{BC}$ in the online phase (a factor of n^2 improvement), and for the offline phase our communication is $O(n^3|C|) + O(n^3|C|) \times \mathsf{BC}$ (a factor of

[1] In $t < n/2$ the recovery is done with a technique called *dispute control* [5], which is repeated n^2 times in the worst case, in contrast to *player elimination*—only suitable for $t < n/3$—which is repeated n times.

n improvement). Table 1 presents our communication in relation to the work of [10], as well as the previous works that achieved G.O.D. with an amount of rounds proportional to $\mathsf{depth}(C)$, independent of n.

Interestingly, for the *online phase*, our protocol matches the *peer-to-peer* communication of the best-known protocol [18], which is linear $O(n|C|)$ (also believed to be optimal [13]).[2] This constitutes significant progress in the direction of matching the communication of G.O.D. protocols for $t < n/2$ with $O(\mathsf{depth}(C))$ rounds, like ours, with protocols that add $O(n^2)$ rounds in the worst case, like [18]. We recall that for $t < n/3$ and perfect security, the task of designing protocols with $O(\mathsf{depth}(C))$ rounds whose communication matches that of $O(\mathsf{depth}(C) + n)$-round protocols was only recently settled in [2].

Remark 1 (On the term $\mathsf{depth}(C) \cdot n^3$*).* Our online communication is $O(n|C| + \mathsf{depth}(C)n^3) + O(n|C| + \mathsf{depth}(C)n^3) \times \mathsf{BC}$. Note that the term $\mathsf{depth}(C)n^3$ is absorbed by $|C|n$, as long as $|C|/\mathsf{depth}(C) = \Omega(n^2)$, in which case the communication becomes $O(|C|n) + O(|C|n) \times \mathsf{BC}$. This can be satisfied for instance if the circuit has uniform width $\Theta(n^2)$, but this is not strictly necessary: for example, a few layers can have a very small amount of multiplication gates (say $o(n)$), while some others may have many more gates (say $\approx n^3$), and the property $|C|/\mathsf{depth}(C) = \Omega(n^2)$ may still be satisfied.

Table 1. Works with G.O.D. for $t < n/2$, and $O(\mathsf{depth}(C))$ rounds (independent of n, even in the worst case). "N/A" in the offline phase means these works did not consider an offline/online separation. We ignore $\mathsf{poly}(n)$ terms that do not depend on C. * The work of [14] does not instantiate the offline phase. The complexity reported is obtained by calculating the cost of generating their preprocessing using our protocol. For this, we take their preprocessing size, $n^2|C|$, and multiply it by the total communication of our protocol, leading to $O(n^3 \cdot n^2|C|)$.

Work	**Offline Comm.**	**Online Comm.**
[24]	N/A	$\omega(\|C\|n^5) + \omega(\|C\|n^5) \times \mathsf{BC}$
[10]	$O(\|C\|n^4) + O(\|C\|n^4) \times \mathsf{BC}$	$O(\|C\|n^3) + O(\|C\|n^3) \times \mathsf{BC}$
[14]	$O(\|C\|n^5) + O(\|C\|n^5) \times \mathsf{BC}$ *	$O(\|C\|n + \mathsf{depth}(C)n^3) + 0 \times \mathsf{BC}$
Ours	$O(\|C\|n^3) + O(\|C\|n^3) \times \mathsf{BC}$	$O(\|C\|n + \mathsf{depth}(C)n^3)+$ $O(\|C\|n + \mathsf{depth}(C)n^3) \times \mathsf{BC}$

[2] Note that our term $O(|C|n) \times \mathsf{BC}$ (which is not present in [18]) would require all parties to *receive* at least $n|C|$ messages, which in practice means a communication of at least $n^2|C|$, widening the gap between the protocol from [18] and ours. See also Remark 1.

1.2 Other Related Work

We have already discussed the works of [5,8,10,11,14,18]—which are the works most related to us—as well as their comparison with respect to our work. We present in the full version a more detailed description of how these protocols work. As other related literature, an important mention is [22], which presents a series of compilers which, among other things, enable "upgrading" secure-with-abort protocols into G.O.D.. Their approach also results in a protocol whose round complexity depends on n, since it consists of identifying corrupt parties and then re-running certain parts of the protocol.

Using information-theoretic randomized encodings [20,21], it is possible to achieve statistically secure MPC in the $n = 2t + 1$ setting with *constant* round complexity [3] (i.e. the round complexity is independent of both n and $\mathsf{depth}(C)$) where the constant is 4. This is akin to how Garbled Circuits (which are an instance of *computationally* secure randomized encoding) enable us to achieve constant round complexity in the computational setting. In such protocols, the communication complexity is always proportional to the size of the randomized encoding. With current known techniques, the size of information-theoretic randomized encoding grows exponentially with the circuit depth and reducing this exponential dependency has been a big open problem for the past two decades. Hence, this approach of using randomized encodings to get low round complexity is currently practical only for NC^1 circuits.

1.3 Overview of Our Techniques

We discuss at a very high level how our final protocol is achieved. First, to avoid the extra n^2 rounds, we must deviate from the dispute-control paradigm used in all communication-efficient works [5,8,18]. Instead, let us take as a starting point the protocol from [10, Section VI], which is more communication heavy but removes the round dependency on n. In [10] the authors make use of the verifiable secret-sharing (VSS) ideas from [11,24], which enable parties to obtain sharings $[s]$ of a secret in such a way that the honest parties can always reconstruct the given secret s, using a *constant* number of rounds. The authors make use of multiplication triples [4], produced in an offline phase. With this preprocessing at hand, the online phase is comprised only of linear combinations and reconstruction of secret-shared data, which is possible thanks to the VSS guarantees. The main contribution of [10] lies in the generation of the multiplication triples. Traditionally, the most efficient approach for triple generation is the first generating so-called double-sharings [12], which can be later used to obtain triples. However, this approach requires more than what VSS can provide: there are certain proofs of correctness that the parties must perform, which ultimately add substantially to the final costs (intuitively, these complexities come from local multiplication increasing the degree from t to $2t$). Instead, the work of [10] introduces a novel approach which only relies on the basic properties of VSS, letting each party contribute with triples directly, which are later checked for correctness and "combined" in such a way that truly random triples are produced.

Intuitively, this only relies on sharing, linear combinations, and reconstructions, and hence can be handled by the underlying VSS alone. Now, this approach is *less* communication-efficient than using double-sharings, but it is much more suitable for reducing the number of rounds. From the above, in order to improve the communication complexity of [10], which maintains the number of rounds we are looking for, it is imperative to improve that of the underlying VSS.

Optimizing—But Weakening—Verifiable Secret-Sharing. We start from the VSS used in [10], which is that from [24] in conjunction with the so-called information-checking protocol (ICP) from [11]. Intuitively, these techniques involve distributing Shamir sharings, and additionally, distributing shares of each share to the parties. The shares-of-shares exist so that, when a party announces a share, it can prove to the others this is correct by also showing them the shares of the announced share, which the parties can contrast with the share-of-share they have internally. Now, a corrupt party may complain of a correctly announced share, and to prevent this the ICP machinery is used: that ensures that (1) honest parties can always prove the correctness of their shares, and (2) corrupt parties who modify their shares are identified. The degree of the polynomials is t, which requires $t+1$ shares to be reconstructed. By using the "signatures", the $\leq t+1$ correct shares coming from the honest shares can be identified, which allows for the reconstruction of the secret. Inspired by [1], our approach to improve the complexity of this VSS is to make use of *packed secret-sharing* [16], which allows for having $\ell \geq 1$ secrets instead of only one, without penalty in the communication costs. However, using packed secret-sharing comes at the expense of increasing the degree of the underlying polynomials from t to $t+(\ell-1)$, and in particular reconstruction now requires more shares than honest parties. This is not a problem in [1], which is set in $t < n/3$, but in our $t < n/2$ regime extra care is needed.

We use packed secret-sharing twice, resulting in packed vectors of dimension $\Theta(n^2)$. First, we use degree $(t+(\ell-1))$ to secret-share $\ell = \Theta(n)$ secrets at once (we will specify the exact value of ℓ), but crucially, we use degree-t for the "shares of shares", which ensures the $t+1$ honest parties alone still have enough joint information to reconstruct the secrets. Now, each share-of-share is signed using the ICP from [11], which at a high level works by secret-sharing the message to be signed towards the parties, so that later on when this message is revealed, the parties can jointly verify if this is consistent with the shares they hold. In [11] degree-t polynomials are used, but a useful observation from [23] is that this can be improved by using packed secret-sharing, signing multiple messages towards multiple verifiers simultaneously. We adapt their ideas to our setting. This requires a batch of $m = \Theta(n)$ shares to be signed, each of which corresponds to $\ell = \Theta(n)$ secrets, so overall our VSS works on vectors of dimension $m\ell = \Theta(n^2)$.

Finally, an important remark is that our construction does not directly instantiate the notion of VSS. Increasing the degree from t to $t+(\ell-1)$ or, as we will see, $t+2(\ell-1)$ in some cases, comes at the expense of corrupt parties being able to disrupt the reconstruction if they decide to misbehave or abort. However, we still guarantee the crucial property that such corrupt parties are identified. We call this weaker notion *detectable secret-sharing* (DSS), and one of our core contributions, on top of the formal definition of such primitive as a UC functionality and its efficient instantiation, is showing that this weaker form of VSS can still be useful for our goal of MPC with $O(\mathsf{depth}(C))$ rounds. We discuss this below.

1.4 Our MPC Protocol

Since our core secret-sharing primitive operates on vectors instead of individual values, we cannot make direct use of the MPC approach from [10], which follows the standard Beaver triple paradigm [4]. Instead, we adapt the ideas from the recent Turbopack work by Escudero et al. [15], which shows how to efficiently make use of packed secret-sharing, supporting arbitrary circuits without noticeable overhead. The details are provided in Sect. 5, but at a high level, there are two main components required in Turbopack. In the offline phase, the main challenge is generating packed multiplication triples, while in the online phase, the main challenge is reconstructing degree-$(t+2(\ell-1))$ secrets. For the first part, we show how to adapt the triple extraction ideas from [10], which quite surprisingly turn out to also work for the packed secret-sharing regime. The most interesting and challenging part is addressing the degree-$(t+2(\ell-1))$ reconstructions.

Recall that in our DSS, the adversary may completely halt the reconstruction of degree-$(t+2(\ell-1))$ secrets. This is because, even though as in [11,24] our scheme guarantees that honest parties can convince the others that their announced shares are correct, and also that corrupt parties announcing incorrect shares are identified, given that there are only $t+1$ honest parties, only $t+1$ out of the $t+2(\ell-1)+1$ shares needed for reconstruction are guaranteed to be announced. That is, there could be $2(\ell-1)$ shares missing. Our core observation is the following. If the t corrupt parties collectively send less than these $2(\ell-1)$ shares, hence halting reconstruction, it is because more than $t-2(\ell-1)$ cheaters misbehaved, and crucially, their identities become known due to the properties of our DSS. At this point these parties can be removed, restarting the computation with threshold $t' < t-(t-2(\ell-1))$ and total number of parties $n' < n-(t-2(\ell-1))$. This time, the corruption ratio is t'/n', and it turns out we can upper bounded it by $1/3$ as long as we take $\ell \leq \frac{n+6}{8}$. The rest of the protocol is now set in the $t' < n'/3$ regime, point in which we can apply any existing work for that threshold to finish the computation. In particular, we can use the recent work of [1], which achieves perfect security but most importantly, requires linear communication and has no overhead in terms of the number of rounds.[3] Note that the adversary can only cause an abort-and-restart *once*, hence keeping the overall number of rounds $O(\mathsf{depth}(C))$.

[3] Furthermore, this protocol can presumably be optimized by avoiding the instantiation of the broadcast channel—which comes "for free" in our setting—and relaxing perfect security to statistical, but we find it to be unnecessary for our feasibility results.

There is a subtle issue when instantiating this novel idea. Restarting the protocol from scratch may allow the parties to change their inputs, which is not secure if in the first run the adversary was able to learn some information about the output. We propose two ways to address this, which perform differently depending on the amount of inputs versus amount of outputs. The first approach is more suitable if there are not many inputs, and consists of having the parties provide sharings of their inputs not only in our DSS scheme—which is packed and does not guarantee reconstruction—but also in the original VSS of [11,24], while proving these two are consistent. In this way, if there is a restart, the parties can provide shares of the inputs for the $t' < n'/3$ protocol, while proving they hold the same secrets as the initial VSS sharings.

The second approach is more adequate if there are not many outputs. First, we note that it is fine to restart the computation while allowing the parties to change their inputs as long as this happens before the output phase, since in this case no sensitive leakage occurs. Now, we modify the output phase as follows: instead of attempting straight reconstruction of the degree-$(t+2(\ell-1))$ output sharings, the parties first convert the packed sharing of the outputs in the main protocol into a non-packed VSS representation. We design this conversion mechanism so that it does not leak anything about the outputs in case it aborts. This way, there are two potential outcomes: either the conversion succeeds, point in which the outputs are VSS'ed and then can be reconstructed with no abort, or the conversion fails, which does not leak anything and hence it is fine to restart the computation with $t' < n'/3$, even if the parties change their inputs. Details on these protocols are given in the full version.

2 Preliminaries

Let κ be a statistical security parameter. We consider a finite field $\mathbb{F}$ with $|\mathbb{F}| = \omega(\mathtt{poly}(\kappa))$, so that $\mathtt{poly}(\kappa)/|\mathbb{F}| = \mathtt{negl}(\kappa)$. Given two strings x and y, we denote by $x\|y$ the concatenation of the two. We denote length-ℓ vectors as $\boldsymbol{v} = (v^1, \ldots, v^\ell)^\intercal \in \mathbb{F}^\ell$. Given two vectors $\boldsymbol{u} = (u^1, \ldots, u^\ell)^\intercal, \boldsymbol{v} = (v^1, \ldots, v^\ell)^\intercal \in \mathbb{F}^\ell$, we define $\boldsymbol{u} * \boldsymbol{v} = (u^1 \cdot v^1, \ldots, u^\ell \cdot v^\ell)^\intercal$ as the component-wise multiplication of $\boldsymbol{u}$ and $\boldsymbol{v}$. We denote $m \times n$ matrices as $\boldsymbol{M} \in \mathbb{F}^\ell$. For any given $C \subseteq [n]$, we denote by $\boldsymbol{M}^C$ the sub-matrix of $\boldsymbol{M}$ corresponding to the columns with indices C. We study MPC amongst n parties in the setting where the number of corrupted parties is exactly t with $n = 2t + 1$. We denote the set of honest parties as $\mathsf{Hon} \subseteq [n]$ and the set of corrupted parties as $\mathsf{Corr} \subseteq [n]$. We say that a protocol has communication complexity $\mathsf{P2P}(M) + N \times \mathsf{BC}(L)$ if it sends M field elements in total over the peer-to-peer channels, and it calls the broadcast channel N times with messages containing L field elements. A degree-d univariate polynomial over $\mathbb{F}$ is of the form $f(x) = \sum_{i=0}^{d} c_i x^i$, where $c_i \in \mathbb{F}$. We say that a collection of field elements $z_{i_1}, \ldots, z_{i_m}$ for $m > d$ and unique $i_1, \ldots, i_m \in \mathbb{F}$ is consistent with a degree-d polynomial, if there exists some degree-d polynomial $f(x)$ such that $f(i_j) = z_{i_j}$ for $i_j \in [m]$. A degree-(d_x, d_y) bivariate polynomial over $\mathbb{F}$ is of the form $F(x, y) = \sum_{i=0}^{d_x} \sum_{j=0}^{d_y} c_{i,j} x^i y^j$ where $c_{i,j} \in \mathbb{F}$. For $i \in \mathbb{F}$, we can isolate univariate polynomials $f_i(x) = F(x, i)$ and $g_i(y) = F(i, y)$ of degree d_x and d_y respectively. In this paper, we will always assume that $d_x = \max\{d_x, d_y\}$. In the full version, we present some basic lemmas regarding bivariate polynomials.

3 Linear Batched Information-Checking Signatures

In this section, we introduce a crucial building block that will be used in our packed detectable secret sharing scheme: linear batched information-checking signatures (IC signatures). This primitive and its construction are based on that of [23], which in turn are based on that of [11,24]. A batched IC signature protocol is executed amongst n parties and allows a dealer D to send a "signature" σ of a batch To ensure that a corrupt INT (or a corrupt D) does not cheat, the n parties, who we call verifiers, each get a "share" of the signature. Importantly, the corrupted parties' shares should together not reveal anything about $\boldsymbol{s}$. Later, INT can reveal the signature σ of $\boldsymbol{s}$ to the n verifiers. Using the shares previously received, the verifiers then decide whether or not to accept the signature. In fact, we allow D to sign many such batches, based on which INT can add their corresponding signatures together, to get a signature of their sum. We also allow INT to compute the signature of a signed batch of secrets component-wise multiplied with some public vector $\boldsymbol{u} \in \mathbb{F}^{\ell}$, using the signature of the original batch.

3.1 IC Signature Ideal Functionality

We now formally introduce our ideal functionality for linear batched IC signatures. The properties that we want from an IC signature are intuitively as follows: (i) If the dealer D is honest, then with all-but-negligible probability, the honest verifiers will only accept a signature σ on the batch $\boldsymbol{s}$ input by D; (ii) If the intermediary INT is honest, then after the signing phase, INT knows a signature σ on some batch $\boldsymbol{s}$ that the honest verifiers will later accept with all-but-negligible probability; and (iii) If both the dealer D and intermediary INT are honest, then nothing about $\boldsymbol{s}$ is revealed to the corrupt verifiers before the reveal phase. Note that if both the dealer D and intermediary INT are corrupted, we guarantee nothing about the signatures, and in fact, INT can decide to reveal a signature σ for any $\boldsymbol{s}$ of the adversary's choosing.

Furthermore, we require the following linearity properties from our IC signatures: (i) Given a signature σ_1 on $\boldsymbol{s}_1$ and a signature σ_2 on $\boldsymbol{s}_2$, we can define a signature σ_3 on $\boldsymbol{s}_1 + \boldsymbol{s}_2$ with the above properties; and (ii) Given a signature σ on $\boldsymbol{s}$ and a public vector $\boldsymbol{u} \in \mathbb{F}^{\ell}$, we can define a signature σ' on $\boldsymbol{s} * \boldsymbol{u}$ with the above properties, only if σ is a linear combination of other signatures that were not themselves multiplied by a public vector.

This latter property of multiplication with a public vector is our main contribution to IC signatures. We now present the ideal functionality $\mathcal{F}_{\mathsf{batch\text{-}IC}}$, which captures the above properties (Note: as far as we are aware, there has been no formal treatment in UC, or any other simulation-based framework, of IC signatures individually in prior works).

Functionality 1: $\mathcal{F}_{\mathsf{batch\text{-}IC}}$

This functionality is parameterized by $\ell \in \mathbb{N}$ and n parties, two of which are the dealer D and intermediary INT. It allows the dealer D to create signatures on several vectors $\boldsymbol{s} \in \mathbb{F}^\ell$, add them together, and multiply them with public vectors $\boldsymbol{u} \in \mathbb{F}^\ell$.

1. In the initialization phase, if either the dealer D or intermediary INT are corrupted, then $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ receives Corr from the adversary. If $\mathsf{Corr} = 1$ then $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ outputs this to all parties and for all future inputs $(\mathsf{sign}, \boldsymbol{s}, \mathsf{sid})$ from D, simply outputs $\boldsymbol{s}$ to all parties, and for all other inputs, ignores them.
2. On input $(\mathsf{sign}, \boldsymbol{s}, \mathsf{sid})$ from the dealer D, where $\boldsymbol{s} \in \mathbb{F}^\ell$, $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ first stores $\mathsf{isMult} \leftarrow 0$ and $\boldsymbol{s}$ with sid, then outputs $(\boldsymbol{s}, \mathsf{sid})$ to INT, and outputs $(\mathsf{signed}, \mathsf{sid})$ to all other parties. Then, if the dealer D is corrupted, $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ receives $\boldsymbol{s}'$ from the adversary, and stores it with sid. Finally, if $\boldsymbol{s}' \neq \boldsymbol{s}$, $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ outputs $(\mathsf{verified}, \boldsymbol{s}', \mathsf{sid})$ to all other parties; otherwise, it outputs $(\mathsf{verified}, \mathsf{sid})$ to all other parties.
3. On input $(\mathsf{reveal}, \mathsf{sid})$ from INT, $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ first sends $\boldsymbol{s}$ stored at sid to the adversary. Then:
   (a) If INT is corrupted and D is honest, $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ asks the adversary whether to reject. If so, it outputs $(\mathsf{reject}, \mathsf{sid})$ to all parties. Otherwise, it outputs $(\boldsymbol{s}, \mathsf{sid})$ to all parties.
   (b) If INT and D are corrupted, $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ asks the adversary whether to reject. If so, it outputs $(\mathsf{reject}, \mathsf{sid})$ to all parties. Otherwise, it receives $\boldsymbol{s}'$ and outputs $(\boldsymbol{s}', \mathsf{sid})$ to all parties.
   (c) Otherwise, $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ outputs $(\boldsymbol{s}, \mathsf{sid})$ to all parties.
4. On input $(\mathsf{add}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3)$ from all parties, let $\mathsf{isMult}_3 \leftarrow 1$, if $\mathsf{isMult}_1 = 1$ or $\mathsf{isMult}_2 = 1$; and $\mathsf{isMult}_3 \leftarrow 0$, otherwise; where $\boldsymbol{s}_1$ and isMult_1 are stored with sid_1 and $\boldsymbol{s}_2$ and isMult_2 are stored with sid_2. $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ stores $\boldsymbol{s}_1 + \boldsymbol{s}_2$ and isMult_3 with sid_3.
5. On input $(\mathsf{mult}, \boldsymbol{u}, \mathsf{sid}, \mathsf{sid}')$ from all parties, where $\boldsymbol{s}$ and $\mathsf{isMult} = 0$ are stored with sid, and $\boldsymbol{u} \in \mathbb{F}^\ell$, $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ stores $\boldsymbol{s} * \boldsymbol{u}$ and $\mathsf{isMult} \leftarrow 1$ with sid'.
6. On input (corr, D) or (corr, INT) from the adversary, $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ sends to the adversary all $(\mathsf{sid}, \boldsymbol{s})$ pairs that have not been revealed yet.

3.2 IC Signature Protocol

Now, we present our protocol $\Pi_{\mathsf{batch\text{-}IC}}$ which instantiates $\mathcal{F}_{\mathsf{batch\text{-}IC}}$. In the initialization phase, the dealer D first sends to each party P_i, random $\alpha_i \leftarrow_\$ \mathbb{F}$, for $i \in [n]$.

Signing. When signing some $\boldsymbol{s} \in \mathbb{F}^\ell$, the dealer D samples a random degree-$(t + \ell - 1)$ polynomial $f(x)$ such that $f(-j + 1) = s^j$ for $j \in [\ell]$ and a random degree-$(t + \ell - 1)$ polynomial $r(x)$. D then sends $f(x)$ and $r(x)$ to INT, and to each other party P_i, $v_i \leftarrow f(\alpha_i)$ and $r_i \leftarrow r(\alpha_i)$. Notice that since $f(x)$ is of degree-$(t+\ell-1)$, an adversary's t points $\{v_j\}_{j \in \mathsf{Corr}}$ reveal nothing about $\boldsymbol{s}$. Now, note that a corrupt D could send $v_i \neq f(\alpha_i)$ to some P_i. In order to catch this bad behavior, INT samples random $\beta \leftarrow_\$ \mathbb{F}$ and broadcasts $(\beta, b(x))$, where $b(x) \leftarrow \beta \cdot f(x) + r(x)$; since β is uniformly random, with all-but-negligible probability, P_i will see that $b(\alpha_i) \neq \beta \cdot v_i + r_i$, and thus D is corrupted. Also, observe that $r(x)$ masks $f(x)$ and thus $\boldsymbol{s}$. However, it could also be the case that D is honest and a corrupted INT broadcasts $(\beta, b(x))$ where $b(x) \neq \beta \cdot f(x) + r(x)$; in this case, the honest D will know that INT is corrupted, and thus the adversary knows $\boldsymbol{s}$, so it can simply broadcast $\boldsymbol{s}$. If D (honest or corrupt) broadcasts $\boldsymbol{s}$, then INT sets the signature $\sigma \leftarrow g(x)$, where $g(x)$ is the degree-ℓ polynomial such that $g(-j+1) = s^j$ for $j \in [\ell]$, and each P_i resets $v_i \leftarrow g(\alpha_i)$. If D does not broadcast $\boldsymbol{s}$, but $b(\alpha_i) \neq \beta \cdot v_i + r_i$ then P_i knows that D is corrupt, and so will accept any signature from INT for this batch of secrets. Also (whether or not D is honest or corrupt), if D does not broadcast $\boldsymbol{s}$, INT sets $\sigma \leftarrow f(x)$.

Adding and multiplication by Public Vectors. To add two signatures together, INT simply sets $\sigma_3(x) \leftarrow \sigma_1(x) + \sigma_2(x)$, and each P_i sets $v_{3,i} \leftarrow v_{1,i} + v_{2,i}$, where it stored $v_{1,i}$ and $v_{2,i}$ for σ_1 and σ_2, respectively. To multiply σ by some public vector $\boldsymbol{u} \in \mathbb{F}^\ell$, let $u(x)$ be the degree-$(\ell-1)$ polynomial such that $u(-j+1) = u^j$ for $j \in [\ell]$. INT simply sets $\sigma'(x) \leftarrow \sigma(x) \cdot u(x)$ (so that it is of degree $t+2\ell-2$) and each P_i sets $v'_i \leftarrow v_i \cdot u(\alpha_i)$.

Revealing. Finally, to reveal a signature, INT simply broadcasts $\sigma(x)$. Then each P_i broadcasts accept if σ is of degree at most $t + 2\ell - 2$ and $\sigma(\alpha_i) = v_i$ or they already marked D as corrupt for this signature (or any of which this signature consists). If at least $t+1$ parties broadcast accept, then the honest parties set $s^j \leftarrow \sigma(= j+1)$ for $j \in [\ell]$ and output $(s^1, \ldots, s^\ell)$; otherwise they output reject. Note that if D is honest and INT is corrupted, for any given honest P_i, if σ is incorrect, then the probability that $\sigma(\alpha_i) = v_i$ is negligible, by the Schwartz-Zippel Lemma, since α_i is random and unknown to the adversary. Thus, with all-but-negligible probability, there will be no P_i such that $\sigma(\alpha_i) = v_i$ for incorrect σ, and thus, the honest parties will only accept a correct σ (since there are at most $t < t+1$ corrupted parties). Observe also that in the case of an honest INT and corrupted D, from above, we will already have that $\sigma(\alpha_i) = v_i$, or P_i marked D as corrupt for this signature, for all $\geq t+1$ honest P_i, and thus all honest P_i will accept.

Rerandomizaing Signatures of Degree-$(2t+2\ell-2)$ *Before Revealing.* One subtlety in the security proof occurs if both D and INT are honest, and a multiplication with some $\boldsymbol{u}$ occurs, boosting the degree of σ to $2t+2\ell-2$. In this case, when σ is revealed in the ideal world, the simulator $\mathcal{S}$ only has at most t corrupted parties' shares and the ℓ points corresponding to the underlying signed $\boldsymbol{s}$, and thus cannot correctly simulate the polynomial $\sigma(x)$. For this reason, before broadcasting σ, INT re-randomizes it with a degree-$(2t+2\ell-2)$ polynomial $o(x)$ given to INT

by D in the initialization phase, such that $o(-j+1)=0$ for $j \in [\ell]$. Each party P_i also adds to v_i, $o_i \leftarrow o(\alpha_i)$, given to them by D in the initialization phase. Similarly to before, for honest INT to ensure that corrupted D gave it $o(x)$ corresponding to the honest parties' o_i values, in the initialization phase it actually receives another such polynomial $o'(x)$ from D, and broadcasts $(\beta, \beta \cdot o(x) - o'(x))$, which the honest parties verifies is consistent with their o_i, o'_i. If an honest INT or D catches a corrupted D or INT, respectively, misbehaving during the initialization phase, then it broadcasts $\mathsf{Corr} \leftarrow 1$. If so, then for all future signatures, D simply broadcasts the secret batch $\boldsymbol{s}$ (since the adversary would learn it anyway).

Now we present the formal protocol $\Pi_{\mathsf{batch\text{-}IC}}$, below.

Protocol 1: $\Pi_{\mathsf{batch\text{-}IC}}$

1. In the initialization phase:
   (a) First, the dealer D samples random α_i for every other party P_i, and sends it to them.
   (b) Then, for $\tau \in [N]$ (in parallel), for some number N:
      i. D samples two random degree-$(t+2\ell-2)$ polynomials $o_1(x)$ and $o_2(x)$ such that $o_1(-j+1)=o_2(-j+1)=0$ for $j \in [\ell]$.
      ii. D then sends $(o_1(x), o_2(x))$ to INT and to each other party P_i, $o_{1,i} \leftarrow o_1(\alpha_i)$ and $o_{2,i} \leftarrow o_2(\alpha_i)$.
      iii. Next, if $o_1(x), o_2(x)$ are not degree-$(t+2\ell-2)$ polynomials such that $o_1(-j+1)=0$ and $o_2(-j+1)=0$ for all $j \in [\ell]$, then INT sets $\mathsf{Corr} \leftarrow 1$ and broadcasts Corr, then all parties output Corr. Otherwise, INT chooses random β and broadcasts $(\beta, o(x))$, where $o(x) \leftarrow \beta \cdot o_1(x) - o_2(x)$.
      iv. D then checks that $o(x) = \beta \cdot o_1(x) - o_2(x)$ and if not, sets $\mathsf{Corr} \leftarrow 1$, and broadcasts Corr, then all parties output Corr.
      v. If D did not broadcast $\mathsf{Corr}=1$ but $o(\alpha_i) \neq \beta \cdot o_1(\alpha_i) - o_2(\alpha_i)$, then P_i sets $\mathsf{dealerbad}_\tau \leftarrow 1$.
      vi. Then INT stores $o_\tau(x) \leftarrow o_1(x)$ and each party P_i stores $o_{\tau,i} \leftarrow o_{1,i}$
   (c) If D or INT broadcasted $\mathsf{Corr}=1$ at any point above, then for all future inputs $(\mathsf{sign}, \boldsymbol{s}_{\mathsf{sid}}, \mathsf{sid})$, the protocol consists of D simply broadcasting $\boldsymbol{s}_{\mathsf{sid}}$, and for all other inputs, the parties ignore them.
2. On input $(\mathsf{sign}, \boldsymbol{s}_{\mathsf{sid}}, \mathsf{sid})$:
   (a) The dealer D samples a random degree-$(t+\ell-1)$ polynomial $f(x)$ such that $f(-j+1)=s^j_{\mathsf{sid}}$ for all $j \in [\ell]$, and a random degree-$(t+\ell-1)$ polynomial $r(x)$.
   (b) D then sends $f(x)$ and $r(x)$ to INT and to each other party P_i, $v_{\mathsf{sid},i} \leftarrow f(\alpha_i)$ and $r_{\mathsf{sid},i} \leftarrow r(\alpha_i)$.
   (c) Next, INT chooses random β and broadcasts $(\beta, b(x))$, where $b(x) \leftarrow \beta \cdot f(x) + r(x)$.
   (d) D then checks that $b(x) = \beta \cdot f(x) + r(x)$ and if not, broadcasts $\boldsymbol{s}$.
   (e) If D indeed broadcasts $\boldsymbol{s}$, let $g(x)$ be the degree-ℓ polynomial such that $g(-j+1)=s_j$ for $j \in [\ell]$. In this case, INT sets $\sigma_{\mathsf{sid}} \leftarrow g(x)$ and each verifier P_i resets their $v_{\mathsf{sid},i} \leftarrow g(\alpha_i)$; otherwise, INT sets $\sigma_{\mathsf{sid}} \leftarrow f(x)$

and each P_i locally sets $\mathsf{dealerbad}_{\mathsf{sid}} \leftarrow 1$ if $\beta \cdot v_{\mathsf{sid},i} + r_{\mathsf{sid},i} \neq b(\alpha_i)$. In both cases, the parties set $\mathsf{isMult}_{\mathsf{sid}} \leftarrow 0$.

3. On input $(\mathsf{reveal}, \mathsf{sid})$:
   (a) (For next available $\tau \in [N]$) *INT* sets $h(x) \leftarrow \sigma_{\mathsf{sid}}(x) + o_\tau(x)$, and each P_i sets $v_{\mathsf{sid},i} \leftarrow v_{\mathsf{sid},i} + o_{\tau,i}$ and $\mathsf{dealerbad}_{\mathsf{sid}} \leftarrow 1$ if $\mathsf{dealerbad}_{\mathsf{sid}}$ was already 1 or $\mathsf{dealerbad}_\tau = 1$.
   (b) *INT* then broadcasts $h(x)$.
   (c) Then, each P_i broadcasts accept if $h(x)$ is degree at most $t+2\ell-2$ and $h(\alpha_i) = v_{\mathsf{sid},i}$, OR $\mathsf{dealerbad}_{\mathsf{sid}} = 1$; otherwise, they broadcast reject.
   (d) If at least $t+1$ parties broadcast accept, then P_i sets $s^j \leftarrow h(-j+1)$ for $j \in [\ell]$ and then outputs $(s^1, \ldots, s^\ell)$; otherwise they output reject.
4. On input $(\mathsf{add}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3)$, *INT* computes $\sigma_{\mathsf{sid}_3} \leftarrow \sigma_{\mathsf{sid}_1} + \sigma_{\mathsf{sid}_2}$. Each other party P_i computes $v_{\mathsf{sid}_3,i} \leftarrow v_{\mathsf{sid}_1,i} + v_{\mathsf{sid}_2,i}$; $\mathsf{dealerbad}_{\mathsf{sid}_3} \leftarrow 1$ if $\mathsf{dealerbad}_{\mathsf{sid}_1} = 1$ or $\mathsf{dealerbad}_{\mathsf{sid}_2} = 1$, otherwise $\mathsf{dealerbad}_{\mathsf{sid}_3} \leftarrow 0$; and $\mathsf{isMult}_{\mathsf{sid}_3} \leftarrow 1$ if $\mathsf{isMult}_{\mathsf{sid}_1} = 1$ or $\mathsf{isMult}_{\mathsf{sid}_2} = 1$, otherwise $\mathsf{isMult}_{\mathsf{sid}_3} \leftarrow 0$.
5. On input $(\mathsf{mult}, \boldsymbol{u}, \mathsf{sid}, \mathsf{sid}')$, the parties first check that $\mathsf{isMult}_{\mathsf{sid}}$ stored with sid satisfies $\mathsf{isMult}_{\mathsf{sid}} = 0$, and abort if not. If so:
   (a) Each party interpolates the degree-$(\ell-1)$ polynomial $u(x)$ such that $u(-j+1) = u^j$ for $j \in [\ell]$.
   (b) *INT* then computes $\sigma_{\mathsf{sid}'} \leftarrow \sigma_{\mathsf{sid}} \cdot u(x)$ and each other party P_i computes $v_{\mathsf{sid}',i} \leftarrow v_{\mathsf{sid},i} \cdot u(\alpha_i)$, $\mathsf{dealerbad}_{\mathsf{sid}'} \leftarrow \mathsf{dealerbad}_{\mathsf{sid}}$, and $\mathsf{isMult}_{\mathsf{sid}} \leftarrow 1$.

Efficiency of $\Pi_{\mathsf{batch\text{-}IC}}$. First, it is clear the initialization phase takes $O(1)$ rounds and costs $\mathsf{P2P}(O(n))$ for sending the α_i. We will count the cost of generating the $o_\tau(x)$ in the corresponding reveal phase below.

In the signing phase, D first sends $f(x), r(x)$ to *INT*, which costs $\mathsf{P2P}(2(t+\ell))$. D then sends the v_i, r_i to the parties P_i, which costs $\mathsf{P2P}(2n)$ values. *INT* then broadcasts $(\beta, b(x))$ which costs $1 \times \mathsf{BC}(1+t+\ell)$. Then, in the worst case, D broadcasts $\boldsymbol{s}$, which costs another $1 \times \mathsf{BC}(\ell)$. Thus, the signing phase costs $\mathsf{P2P}(O(n+\ell))$ and $1 \times \mathsf{BC}(O(n+\ell))$. If $\ell = \Theta(n)$, then this is $\mathsf{P2P}(O(n))$ and $1 \times \mathsf{BC}(O(n))$. It is clear that the signing phase takes $O(1)$ rounds. Note that both adding and multiplication are local operations.

In the reveal phase, *INT* broadcasts $h(x)$ which costs at most $1 \times \mathsf{BC}(t+2\ell-2)$. Then, each P_i broadcasts accept or reject which costs $n \times \mathsf{BC}(1)$. Additionally, in the initialization phase, D sends $o_1(x), o_2(x)$ to *INT*, the former of which *INT* adds to $\sigma(x)$ to get $h(x)$, which costs $\mathsf{P2P}(2(t+2\ell-2))$, and $o_{1,i}, o_{2,i}$ to each P_i, which costs $\mathsf{P2P}(2n)$. Then *INT* broadcasts $(\beta, o(x))$, which costs $1 \times \mathsf{BC}(t+2\ell-1)$. Counting the cost to generate $o_1(x)$ as part of the reveal phase cost, we get the reveal phase costs $\mathsf{P2P}(O(n+\ell))$, $O(n) \times \mathsf{BC}(O(1))$, and $O(1) \times \mathsf{BC}(O(n+\ell))$. If $\ell = \Theta(n)$, then this is $\mathsf{P2P}(O(n))$, $O(n) \times \mathsf{BC}(O(1))$, and $O(1) \times \mathsf{BC}(O(n))$. It is clear that the reveal phase takes $O(1)$ rounds.

Theorem 1. *$\Pi_{\mathsf{batch\text{-}IC}}$ UC-realizes $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ for any $\ell = \mathtt{poly}(\kappa)$ with probability $1 - \mathtt{negl}(\kappa)$.*

The proof is in the full version.

4 Packed, Batched, (Mass) Detectable Secret Sharing

In this section, we introduce our packed, batched, (mass) detectable secret sharing (DSS) ideal functionality and protocol. We base the protocol off of that of [11] and take from [1] the idea of "packing" many secrets into a single bivariate polynomial, as well as batching many bivariate polynomials to amortize costs. A DSS protocol is executed amongst n parties and allows any given P_i to act as a dealer and secret share a batch of secret vectors $\boldsymbol{s}_1, \dots, \boldsymbol{s}_m \in \mathbb{F}^\ell$. As usual, we want the corrupted parties' shares to reveal nothing about $\boldsymbol{s}_1, \dots, \boldsymbol{s}_m \in \mathbb{F}^\ell$. Later, the parties can choose to publicly reconstruct $\boldsymbol{s}_1, \dots, \boldsymbol{s}_m$; the reconstruction must either succeed, or $\Omega(n)$ corrupted parties are publicly identified (the exact number depends on ℓ). In fact, we allow the parties to add their shares of many such sharings together, which results in a sharing of the vector-wise sum of the underlying batches of secret vectors. We also allow the parties to compute a sharing of a shared batch of secrets, element-wise and component-wise multiplied by some public vectors $\boldsymbol{u}_1, \dots, \boldsymbol{u}_m \in \mathbb{F}^\ell$, using the original sharing.

4.1 Detectable Secret Sharing Ideal Functionality

We now present our packed, batched, DSS ideal functionality. The properties that we want from a DSS are as follows: (i) If the given dealer P_i is honest, then all honest parties will complete the sharing phase; (ii) If the given dealer P_i is honest, then nothing about $\boldsymbol{s}_1, \dots, \boldsymbol{s}_m$ are revealed before the reconstruction phase; and (iii) If all honest parties finish a sharing phase, then there exists fixed $\boldsymbol{x}_1, \dots, \boldsymbol{x}_m$ such that (a) if the given dealer P_i is honest, then each $\boldsymbol{x}_i = \boldsymbol{s}_i$, and (b) if all honest parties start the reconstruction phase, either it succeeds with them outputting $\boldsymbol{x}_1, \dots, \boldsymbol{x}_m$, or it fails, but $\Omega(n)$ corrupted parties are identified.

Furthermore, we require the following linearity properties from our DSS: (i) Given a sharing of $\boldsymbol{s}_{1,1}, \dots, \boldsymbol{s}_{1,m}$ and a sharing of $\boldsymbol{s}_{2,1}, \dots, \boldsymbol{s}_{2,m}$, the parties can compute a sharing of $\boldsymbol{s}_{1,1} + \boldsymbol{s}_{2,1}, \dots, \boldsymbol{s}_{1,m} + \boldsymbol{s}_{2,m}$ with the above properties; and (ii) Given a sharing of $\boldsymbol{s}_1, \dots, \boldsymbol{s}_m$ and public vectors $\boldsymbol{u}_1, \dots, \boldsymbol{u}_m \in \mathbb{F}^\ell$, the parties can compute a sharing of $\boldsymbol{s}_1 * \boldsymbol{u}_1, \dots, \boldsymbol{s}_m * \boldsymbol{u}_m$ with the above properties, only if $\boldsymbol{s}_1, \dots, \boldsymbol{s}_m$ is a linear combination of other sharings that were not themselves multiplied by a batch of public vectors.

Our main contributions to DSS are using packing and batching in the statistical setting, $t < n/2$ setting to amortize costs, as well as the mass detectability property in case of failure of reconstruction. Now we present the ideal functionality $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$, which captures the above properties.

Functionality 2: $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$

This functionality is parameterized by $\ell, m \in \mathbb{N}$ and n parties. It allows parties to create several size-m batches of packed Verifiable Secret Sharings of size-ℓ vectors, add them together, and multiply them with size-m batches of size-ℓ public vectors.

1. On input $(\mathsf{share}, (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m), \mathsf{sid})$ from party P_i, where each $\boldsymbol{s}_j \in \mathbb{F}^\ell$, and $(\mathsf{share}, P_i, \mathsf{sid})$ from all honest parties, $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ first sets $\mathsf{isMult} \leftarrow 0$. Then if P_i is corrupted, $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ first asks the adversary whether to continue. If so, $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ stores $(P_i, \mathsf{isMult}, (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m))$ with sid; else, $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ outputs abort to all parties. If P_i is honest, $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ stores $(P_i, \mathsf{isMult}, (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m))$ with sid.
2. On input $(\mathsf{reconstruct}, \mathsf{sid})$ from all parties, where $(\cdot, \mathsf{isMult}, (\boldsymbol{s}_1 \ldots, \boldsymbol{s}_m))$ is stored at sid, $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ first sends $(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m)$ to the adversary and asks whether to continue. If so, $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ outputs $(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m)$ to all parties. Otherwise, $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ receives from the adversary a set of indices $T \subsetneq [n]$ corresponding to corrupted parties such that $|T| > t - \ell + 1$ if $\mathsf{isMult} = 0$ and $|T| > t - 2\ell + 2$ if $\mathsf{isMult} = 1$, and outputs T to all parties.[a]
3. On input $(\mathsf{add}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3)$ from all parties, where $(\cdot, \mathsf{isMult}_1, (\boldsymbol{s}_{1,1} \ldots, \boldsymbol{s}_{1,m}))$ is stored with sid_1 and $(\cdot, \mathsf{isMult}_2, (\boldsymbol{s}_{2,1} \ldots, \boldsymbol{s}_{2,m}))$ is stored with sid_2, $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ stores $(\bot, \mathsf{isMult}_3, (\boldsymbol{s}_{1,1} + \boldsymbol{s}_{2,1}, \ldots, \boldsymbol{s}_{1,m} + \boldsymbol{s}_{2,m}))$ with sid_3, where $\mathsf{isMult}_3 \leftarrow 0$ if $\mathsf{isMult}_1 = \mathsf{isMult}_2 = 0$, and $\mathsf{isMult}_3 \leftarrow 1$ otherwise.
4. On input $(\mathsf{mult}, (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m), \mathsf{sid}, \mathsf{sid}')$ from all parties, where each $\boldsymbol{u}_i \in \mathbb{F}^\ell$ and $(\cdot, 0, (\boldsymbol{s}_1 \ldots, \boldsymbol{s}_m))$ is stored at sid, $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ stores $(\bot, 1, (\boldsymbol{s}_1 * \boldsymbol{u}_1, \ldots, \boldsymbol{s}_m * \boldsymbol{u}_m))$ with sid'.
5. On input (corr, P_i) from the adversary, $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ sends to the adversary all pairs $(\mathsf{sid}, (P_i, (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m)))$.

[a] If $\ell = 1$, $(\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m)$ will always be output to the parties since there are only t corrupted parties and thus the adversary cannot send T such that $|T| > t$.

4.2 Detectable Secret Sharing Subroutines

Before presenting our DSS protocol $\Pi_{\mathsf{Packed\text{-}DSS}}$, we will first present various procedures which $\Pi_{\mathsf{Packed\text{-}DSS}}$ uses. Note, however, that $\Pi_{\mathsf{Packed\text{-}DSS}}$ starts with each party initializing separate instances of $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ as a dealer with each other party acting as intermediary. The procedures will use these instances of $\mathcal{F}_{\mathsf{batch\text{-}IC}}$.

Sharing Procedure $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$. We begin by presenting the sharing procedure, $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}(d_x, \mathsf{sid}, (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m))$, below. When a party P_i wants to secret share a batch of vectors $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m \in F^\ell$,[4] with degree $n-1 \geq d_x \geq t+\ell-1$, it begins by sampling m random degree-(d_x, t) bivariate polynomials such that $F_\eta(-l+1, 0) = s^l_\eta$ for $l \in [\ell], \eta \in [m]$. Then, letting $z^{jk}_\eta \leftarrow F_\eta(j,k)$ and $\boldsymbol{z}^{jk}_\eta \leftarrow (z^{jk}_1, \ldots, z^{jk}_m)$, P_i invokes the $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ instance with P_j as intermediary on input $\boldsymbol{z}^{jk}_\eta$ and $\boldsymbol{z}^{kj}_\eta$ (thus implicitly sending to P_j these vectors), for $j, k \in [n]$. Each P_j then ensures that the points it receives define valid degree-d_x and degree-$(t+\ell-1)$ polynomials, respectively. If not, it reveals their points to all of the parties, using $\mathcal{F}_{\mathsf{batch\text{-}IC}}$. Then, if a party sees such a set of bad points from some

[4] For a given instance of $\Pi_{\mathsf{Packed\text{-}DSS}}$, we use the same packing parameter ℓ and batching parameter m for each call to $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$.

other party P_k (checking that indeed, the points are bad), it aborts. Then, P_j invokes the $\mathcal{F}_{\text{batch-IC}}$ instance with P_k as intermediary on input $\boldsymbol{z}_\eta^{kj}$ (thus implicitly sending to P_k this vector), for $k \in [n]$. Next, P_j compares those points it received from P_k to those received from the dealer P_i, and if there is any inconsistency, reveals those points that P_i gave it to all of the parties, using $\mathcal{F}_{\text{batch-IC}}$. Then, P_j checks if some P_k revealed points that are not consistent with those it received from P_i, and if so, reveals those points that P_i gave it to all of the parties, using $\mathcal{F}_{\text{batch-IC}}$. If any pair of parties $P_j \neq P_k$ revealed two different vectors of points from the dealer P_i, then all parties abort. Otherwise, each party P_j outputs its share, $(\boldsymbol{z}_{\mathsf{sid}}^{j1}, \ldots, \boldsymbol{z}_{\mathsf{sid}}^{jn})$.

We will only ever explicitly use $\pi_{\text{Packed-DSS-Share}}$ in the following to generate sharings with degree $d_x = t + \ell - 1$ or degree $d_x = t + (2\ell - 1)$. If the parties do not abort in $\pi_{\text{Packed-DSS-Share}}$, we denote a sharing of $\boldsymbol{s} = (\boldsymbol{s_1}, \ldots, \boldsymbol{s_m})$ with degree $d_x = t + \ell - 1$ as $[\![\boldsymbol{s}]\!]$ and a sharing with degree $d_x = t + \ell - 1$ as $[\![\boldsymbol{s}]\!]_*$.

Procedure 2: $\pi_{\text{Packed-DSS-Share}}(P_i, d_x, \mathsf{sid}, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_m)$

This procedure takes in the party P_i acting as dealer, $n - 1 \geq d_x \geq t + \ell - 1$ and produces a degree-(d_x, t) sharing of $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m$.

1. Party P_i samples m random bivariate polynomials $F_1(x, y), \ldots, F_m(x, y)$ of degree at most d_x in x and t in y, such that $F_\eta(-l + 1, 0) = s_\eta^l$ for $l \in [\ell], \eta \in [m]$.
2. Let $z_\eta^{jk} \leftarrow F_\eta(j, k)$ and $\boldsymbol{z}_{\mathsf{sid}}^{jk} \leftarrow (z_1^{jk}, \ldots, z_m^{jk})$. P_i invokes the $\mathcal{F}_{\text{batch-IC}}$ instance with P_j as intermediary on inputs $(\mathsf{sign}, \boldsymbol{z}_{\mathsf{sid}}^{jk}, \mathsf{sid}\|k\|0)$ and $(\mathsf{sign}, \boldsymbol{z}_{\mathsf{sid}}^{kj}, \mathsf{sid}\|k\|1)$, for $j, k \in [n]$.
3. Each party P_j checks that for each $\eta \in [m]$, $z_\eta^{j1}, \ldots, z_\eta^{jn}$ received from $\mathcal{F}_{\text{batch-IC}}$ define a degree-t polynomial and $z_\eta^{1j}, \ldots, z_\eta^{nj}$ received from $\mathcal{F}_{\text{batch-IC}}$ define a degree-d_x polynomial. If not, P_j reveals $\boldsymbol{z}_{\mathsf{sid}}^{jk}, \boldsymbol{z}_{\mathsf{sid}}^{kj}$ to all of the parties by invoking $\mathcal{F}_{\text{batch-IC}}$ on $(\mathsf{reveal}, \mathsf{sid}\|k\|0)$ and $(\mathsf{reveal}, \mathsf{sid}\|k\|1)$, for all $k \in [n]$.
4. If a party sees polynomial evaluations from some other party P_k that do not define degree-t or degree-d_x polynomials, respectively, it aborts.
5. Each P_j invokes the $\mathcal{F}_{\text{batch-IC}}$ instance with P_k as the intermediary on input $(\mathsf{sign}, \boldsymbol{z}_{\mathsf{sid}}^{kj}, \mathsf{sid})$, for $k \in [n]$.
6. P_j compares the values $\boldsymbol{z}_{\mathsf{sid}}^{jk}$ which he received from $\mathcal{F}_{\text{batch-IC}}$ for each $k \in [n]$ in the previous round to the values received from P_i. If there is any inconsistency, P_j reveals $\boldsymbol{z}_{\mathsf{sid}}^{jk}$ received from P_i to all of the parties by invoking $\mathcal{F}_{\text{batch-IC}}$ on $(\mathsf{reveal}, \mathsf{sid}\|k\|0)$.
7. P_j checks if some P_k revealed a value $\boldsymbol{z}_{\mathsf{sid}}^{kj}$ which is different from that which P_i gave it. If so, then P_j reveals $\boldsymbol{z}_{\mathsf{sid}}^{kj}$ to all parties by invoking $\mathcal{F}_{\text{batch-IC}}$ on $(\mathsf{reveal}, \mathsf{sid}\|k\|1)$.
8. If for any index pair $(j, k) \in [n] \times [n]$, a party sees two different vectors of points from P_i, then it aborts; otherwise, each party P_j outputs its share $(\boldsymbol{z}_{\mathsf{sid}}^{j1}, \ldots, \boldsymbol{z}_{\mathsf{sid}}^{jn})$.

Now, we prove the following simple lemma about $\pi_{\text{Packed-DSS-Share}}$:

Lemma 1. *If the dealer P_i is honest in $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$, then all honest parties finish $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$ without aborting.*

The proof is in the full version.

Next, we prove the following lemma, which shows that if the values $\boldsymbol{z}_{\mathsf{sid}}^{kj}$ which the honest parties P_j input to $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ in step 5 of $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$, and which thus become part of P_k's share, define degree-t polynomials $g_k(y)$, then they uniquely define the underlying degree-(d_x, t) bivariate polynomials $F_\eta(x, y)$ and thus the shared secrets $\boldsymbol{s}_\eta$.

Lemma 2. *For any $K \subseteq [n]$ such that $|K| \geq d_x + 1$, let $\boldsymbol{z}_{\mathsf{sid}}^{kj}$, for $k \in K$, be the vectors that the honest parties P_j input to $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ in step 5 of $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$. Assume that $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$ does not abort and that for each $\eta \in [m], k \in K$, $\{z_\eta^{kj}\}_{j\in[\mathsf{Hon}]}$ define degree-t polynomials. Then for all $\eta \in [m]$, the $\{z_\eta^{kj}\}_{j\in\mathsf{Hon}}$ for $k \in K$ together define unique degree-(d_x, t) bivariate polynomials $F_\eta(x, y)$.*

The proof is in the full version.

Efficiency of $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$. For analyzing the communication complexity of $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$, we will utilize the efficiency of our $\Pi_{\mathsf{batch\text{-}IC}}$ protocol for $\mathcal{F}_{\mathsf{batch\text{-}IC}}$. P_i, for each P_j, signs length-m vectors $\boldsymbol{z}_{\mathsf{sid}}^{jk}, \boldsymbol{z}_{\mathsf{sid}}^{kj}$ for $k \in [n]$, with $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ which costs $\mathsf{P2P}(O(n^2 \cdot (n+m)))$ and $n^2 \times \mathsf{BC}(O(n+m))$, using $\Pi_{\mathsf{batch\text{-}IC}}$. Then, in the worst case, each P_j could reveal those signatures, which costs $\mathsf{P2P}(O(n^2 \cdot (n + m)))$, $n^3 \times \mathsf{BC}(O(1))$, and $n^2 \times \mathsf{BC}(O(n+m))$, using $\Pi_{\mathsf{batch\text{-}IC}}$. Next, each P_i sends signs for each P_k length-m vectors $\boldsymbol{z}_{\mathsf{sid}}^{kj}$ with $\mathcal{F}_{\mathsf{batch\text{-}IC}}$, which costs $\mathsf{P2P}(n^2 \cdot (n+m))$ and $O(n^2) \times \mathsf{BC}(O(n + m))$, using $\Pi_{\mathsf{batch\text{-}IC}}$. Next, in the worst case, each P_j could reveal the signatures from P_i for all $k \in [n]$, which costs $\mathsf{P2P}(O(n^2 \cdot (n + m)))$, $n^3 \times \mathsf{BC}(1)$, and $n^2 \times \mathsf{BC}(O(n + m))$, using $\Pi_{\mathsf{batch\text{-}IC}}$. Altogether, this is $\mathsf{P2P}(O(n^3 + n^2 m))$, $O(n^3) \times \mathsf{BC}(O(1))$, and $O(n^2) \times \mathsf{BC}(O(n+m))$. If $m = \Theta(n)$, this is $\mathsf{P2P}(O(n^3))$, $O(n^3) \times \mathsf{BC}(O(1))$, and $O(n^2) \times \mathsf{BC}(O(n))$. It is clear that $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$ takes $O(1)$ rounds.

Adding Packed, Batched DSS's and Multiplying Them by Public Vectors. Let us assume that the parties have a packed, batched DSS $[\![\boldsymbol{s}_1]\!]$ and a packed, batched DSS $[\![\boldsymbol{s}_2]\!]$. Adding two such sharings together is a simple, local procedure, $\pi_{\mathsf{Packed\text{-}DSS\text{-}Add}}$, which consists of parties simply adding (the signatures on) their shares together:

Procedure 3: $\pi_{\mathsf{Packed\text{-}DSS\text{-}Add}}([\![\boldsymbol{s}_1]\!], [\![\boldsymbol{s}_2]\!])$

Let $(\boldsymbol{z}_{\mathsf{sid}_1}^{j1}, \ldots, \boldsymbol{z}_{\mathsf{sid}_1}^{jn})$ and $(\boldsymbol{z}_{\mathsf{sid}_2}^{j1}, \ldots, \boldsymbol{z}_{\mathsf{sid}_2}^{jn})$ be party P_j's respective shares of sharings sid_1 and sid_2.

1. For each $j \in [n]$, P_j sets $\boldsymbol{z}_{\mathsf{sid}_3}^{jk} \leftarrow \boldsymbol{z}_{\mathsf{sid}_1}^{jk} + \boldsymbol{z}_{\mathsf{sid}_2}^{jk}$ and all parties invoke the $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ instance with P_j as intermediary and P_k as dealer on input $(\mathsf{add}, \mathsf{sid}_1, \mathsf{sid}_2, \mathsf{sid}_3)$, for $k \in [n]$.[a]
2. Each P_j outputs new shares $(\boldsymbol{z}_{\mathsf{sid}_3}^{j1}, \ldots, \boldsymbol{z}_{\mathsf{sid}_3}^{jn})$.

[a] Note that for our $\Pi_{\text{batch-IC}}$, the add operation is indeed local.

We denote this as $[\![\boldsymbol{s}_3]\!] \leftarrow [\![\boldsymbol{s}_1]\!] + [\![\boldsymbol{s}_2]\!]$. Note that this also works for sharings $[\![\boldsymbol{s}_1]\!]_*$ and/or $[\![\boldsymbol{s}_2]\!]_*$ of higher degree ($d_x = t + 2(\ell - 1)$), in which case we denote $[\![\boldsymbol{s}_3]\!]_*$ as the resulting sharing.

Now, let us assume that the parties have a single packed, batched DSS of secrets $[\![\boldsymbol{s}]\!]$ (note that for such a sharing, $d_x = t + \ell - 1 \leq n - \ell$), and some public vectors $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m \in \mathbb{F}^\ell$. Multiplying the sharing by this batch of public vectors is a simple, local procedure, $\pi_{\text{Packed-DSS-Mult}}$, which consists of parties simply multiplying their shares (and the signatures on those shares) by the degree-$(\ell - 1)$ polynomials $u_\eta(x)$ such that $u_\eta(-l+1) = u_\eta^l$ for $\eta \in [n], l \in [\ell]$:

Procedure 4: $\pi_{\text{Packed-DSS-Mult}}([\![\boldsymbol{s}]\!]_{d_x,t}, (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m))$

Let $(\boldsymbol{z}_{\mathsf{sid}}^{j1}, \ldots, \boldsymbol{z}_{\mathsf{sid}}^{jn})$ be party P_j's shares of sharing sid.

1. Each party first interpolates the degree-$(\ell-1)$ polynomials $u_\eta(x)$ such that $u_\eta(-l+1) = u_\eta^l$ for $\eta \in [m], l \in [\ell]$.
2. Then, for each $j \in [n]$, P_j locally computes $\boldsymbol{z}_{\mathsf{sid}'}^{jk} \leftarrow \boldsymbol{z}_{\mathsf{sid}}^{jk} * (u_1(j), \ldots, u_m(j))$ and all parties invoke the $\mathcal{F}_{\text{batch-IC}}$ instance with P_j as intermediary and P_k as dealer on input $(\mathsf{mult}, (u_1(j), \ldots, u_m(j)), \mathsf{sid}, \mathsf{sid}')$, for $k \in [n]$.[a]
3. Finally, each P_j outputs new shares $(\boldsymbol{z}_{\mathsf{sid}'}^{j1}, \ldots, \boldsymbol{z}_{\mathsf{sid}'}^{jn})$.

[a] Note that for our $\Pi_{\text{batch-IC}}$, the mult operation is indeed local.

We denote this as $[\![\boldsymbol{s}]\!]_* \leftarrow [\![\boldsymbol{s}]\!] * \boldsymbol{u}$, since the new sharing has degree $d_x = t + 2(\ell - 1)$.

Let $F_{\mathsf{sid}',\eta}(x, y)$ be the unique polynomials defined by the $\{z_{\mathsf{sid}',\eta}^{kj}\}_{j \in \mathsf{Hon}}$ for $k \in K$, of some sharings $[\![\boldsymbol{s}']\!]$ output by $\pi_{\text{Packed-DSS-Share}}$, according to Lemma 2. We can again prove the following lemma similar to Lemma 2, which essentially says that for any sharing $[\![\boldsymbol{s}]\!]$ (or $[\![\boldsymbol{s}]\!]_*$) formed by running the addition and multiplication procedures above on sharings $[\![\boldsymbol{s}']\!]$ originally output by $\pi_{\text{Packed-DSS-Share}}$, the $\{\boldsymbol{z}_{\mathsf{sid}}^{jk}\}_{j \in \mathsf{Hon}}$ part of each P_k's share (defined by the corresponding signatures), together uniquely define the underlying degree-(d_x, t) bivariate polynomials $F_\eta(x, y)$, which are equal to the polynomials that result from applying the same addition and multiplication procedures on the $F_{\mathsf{sid}',\eta}(x, y)$ above from the original sharings $[\![\boldsymbol{s}']\!]$.[5] The proof is in the full version.

Lemma 3. *Let the sharing $[\![\boldsymbol{s}]\!]$ (resp. $[\![\boldsymbol{s}]\!]_*$), be the result of addition and multiplication procedures on sharings $[\![\boldsymbol{s}']\!]$ originally output by $\pi_{\text{Packed-DSS-Share}}$. For any $K \subseteq [n]$ such that $|K| \geq d_x + 1$, let $\{\boldsymbol{z}_{\mathsf{sid}}^{kj}\}_{j \in \mathsf{Hon}}$ be the part of each P_k's shares of $[\![\boldsymbol{s}]\!]$ (resp. $[\![\boldsymbol{s}]\!]_*$) defined by the $\mathcal{F}_{\text{batch-IC}}$ instance with P_j as dealer and*

[5] A 'multiplication procedure' multiplying a sharing by $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m$ corresponds to multiplying the polynomials defined by the sharing by the degree-$(\ell-1)$ polynomials $u_1(x), \ldots, u_m(x)$ defined by the above vectors.

P_k as intermediary. Assume that for each $\eta \in [m], k \in K$, $\{z^{kj}_{\mathsf{sid},\eta}\}_{j\in[\mathsf{Hon}]}$ define degree-t polynomials. Then for all $\eta \in [m]$, the $\{z^{kj}_{\mathsf{sid},\eta}\}_{j\in\mathsf{Hon}}$ for $k \in K$ together define unique degree-(d_x, t) bivariate polynomials $F_{\mathsf{sid},\eta}(x, y)$ which are equal to the polynomials which result from applying the same addition and multiplication procedures on the unique polynomials $F_{\mathsf{sid}',\eta}(x, y)$ defined by the $\{z^{kj}_{\mathsf{sid}',\eta}\}_{j\in\mathsf{Hon}}$ for $k \in K$, by Lemma 2.

Essentially, the above lemma shows that our add and multiplication procedures have the desired outcome of performing the corresponding operations. The following corollary will help us show that the correct secrets can then be reconstructed from such sharings. The proof is in the full version.

Corollary 1. *If for each $\eta \in [n], k \in K$, $\{z^{ki}_{\mathsf{sid},\eta}\}_{i\in[n]}$ in P_k's share of $[\![\boldsymbol{s}]\!]$ (resp. $[\![\boldsymbol{s}]\!]_*$) define a degree-t polynomial, then given any $I \subseteq [n]$ such that $|I| = t + 1$ (such as $[t + 1]$) $\{z^{ki}_{\mathsf{sid},\eta}\}_{i\in I}$ can be used to interpolate the same unique degree-(d_x, t) bivariate polynomials $F_{\mathsf{sid},\eta}(x, y)$ as in Lemma 3.*

Reconstruction Procedure $\pi_{\textsf{Packed-DSS-Rec}}$ Next, we present the reconstruction procedure, $\pi_{\textsf{Packed-DSS-Rec}}([\![\boldsymbol{s}]\!])$, which the honest parties use to reconstruct the batch of secret vectors defined by their shares of the sharing $[\![\boldsymbol{s}]\!]$. All parties P_k first reveal their share $\boldsymbol{z}^{k1}_{\mathsf{sid}}, \ldots, \boldsymbol{z}^{kn}_{\mathsf{sid}}$ to all parties using $\mathcal{F}_{\textsf{batch-IC}}$. Then, each P_k checks if the points that each P_j revealed define degree-t polynomials in y, and if not, marks them as corrupt. Then, if the number of parties marked corrupt is greater than $2t - d_x$, the honest parties output those parties' identities that are marked corrupt (note that sharings must satisfy $d_x \leq n - 1$, so $2t - d_x > 0$). Otherwise, the parties use the shares of those parties that are not marked corrupt to interpolate the unique, correct $F_\eta(x, y)$ (that exist by Corollary 1) and output the corresponding secrets $\boldsymbol{s}_\eta$. Note that this procedure works in exactly the same way for sharings $[\![\boldsymbol{s}]\!]_*$ of higher degree $d_x = t + 2(\ell - 1)$.

Procedure 5: $\pi_{\textsf{Packed-DSS-Rec}}([\![\boldsymbol{s}]\!])$

This procedure takes as input the party's shares of $[\![\boldsymbol{s}]\!]$ of degree $d_x = t + \ell - 1$ or $[\![\boldsymbol{s}]\!]_*$ of degree $d_x = t + 2(\ell - 1)$.

1. Every party P_k reveals $\boldsymbol{z}^{k1}_{\mathsf{sid}}, \ldots, \boldsymbol{z}^{kn}_{\mathsf{sid}}$ to all other parties by invoking for $j \in [n]$, the $\mathcal{F}_{\textsf{batch-IC}}$ instance with P_j as intermediary on input $(\mathsf{reveal}, \mathsf{sid})$.
2. P_k first sets $T \leftarrow \emptyset$ then checks whether P_j's shares revealed in the previous step define degree-t polynomials $F_\eta(j, y)$, $\eta \in [m]$. If not, then P_j is added to T and thus marked as corrupt.
3. If the number of parties marked corrupt in T is greater than $2t - d_x$, then output T and abort.
4. For every P_k not marked as corrupt in $K = [n] \setminus T$, P_j uses the values $F_\eta(k, 1), \ldots, F_\eta(k, t + 1)$, together, to interpolate the unique degree-(d_x, t) bivariate polyonmial $F_\eta(x, y)$.
5. P_j finally outputs $s^l_\eta \leftarrow F_\eta(-l + 1, 0)$ for $\eta \in [m], l \in [\ell]$.

We now have the following lemma, which shows that if $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m$ are output by the honest parties, then they are the correct secrets corresponding to $[\![\boldsymbol{s}]\!]$; otherwise, each party $P_k \in T$ is actually corrupt. The latter is because for sharings output by $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$, honest parties' shares are always consistent with degree-t polynomials, for otherwise they would have aborted. Furthermore, addition and multiplication operations do not affect the degree in the y variable, so the shares always stay consistent with degree-t polynomials. The proof appears in the full version.

Lemma 4. *Let $\{\boldsymbol{z}_{\mathsf{sid}}^{kj}\}_{k \in K, j \in [n]}$ be the points that are revealed via $\mathcal{F}_{\mathsf{batch\text{-}IC}}$ in $\pi_{\mathsf{Packed\text{-}DSS\text{-}Rec}}$. If $|T| \leq 2t - d_x$, then the honest parties output the correct secrets $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m$ defined by the unique degree-$(t + \ell - 1, t)$ bivariate polynomials $(F_1(x, y), \ldots, F_m(x, y))$ from Lemma 3. If $|T| > 2t - d_x$, then the honest parties output (abort, T) such that for each $P_j \in T$, $P_j \in \mathsf{Corr}$.*

Efficiency of $\pi_{\mathsf{Packed\text{-}DSS\text{-}Rec}}$. For analyzing the communication complexity of $\pi_{\mathsf{Packed\text{-}DSS\text{-}Rec}}$, we will utilize the efficiency of our $\Pi_{\mathsf{batch\text{-}IC}}$ protocol for $\mathcal{F}_{\mathsf{batch\text{-}IC}}$. Each P_k simply reveals $\boldsymbol{z}_{\mathsf{sid}}^{kj}$, for $j \in [n]$ with $\mathcal{F}_{\mathsf{batch\text{-}IC}}$, which costs $\mathsf{P2P}(O(n^2 \cdot (n+m)))$, $n^3 \times \mathsf{BC}(1)$, and $n^2 \times \mathsf{BC}(O(n+m))$, using $\Pi_{\mathsf{batch\text{-}IC}}$. If $m = \Theta(n)$, this is $\mathsf{P2P}(O(n^3))$, $O(n^3) \times \mathsf{BC}(O(1))$, and $O(n^2) \times \mathsf{BC}(O(n))$. It is clear that $\pi_{\mathsf{Packed\text{-}DSS\text{-}Rec}}$ takes $O(1)$ rounds.

Creating Random Sharings $[\![\mathbf{0}]\!]_*$. After multiplying sharings by batches of public vectors, the degree of the sharing increases by $\ell - 1$ in x. Thus, in order to securely open such sharings, we need to mask them by random sharings $[\![\mathbf{0}]\!]_*$, of degree $x_x = t + 2(\ell - 1)$, since all sharings created as part of the $\Pi_{\mathsf{Packed\text{-}DSS}}$ protocol will start as degree $d_x = t + \ell - 1$. We use the typical random extraction technique from [13] to do this efficiently, which consists of each party creating their own such random sharings, and then using some super-invertible $(n-t) \times n$ matrix $\boldsymbol{M}$ to take linear combinations of these sharings and then output the resulting sharings that are random to the adversary.

However, it may be that the underlying secrets that corrupted parties share are not equal to $\mathbf{0}, \ldots, \mathbf{0}$. For this, we adapt a standard technique, which takes as input two sharings from the same party which supposedly share $\mathbf{0}, \ldots, \mathbf{0}$, take a random linear combination of the two, then open them to check if they are indeed sharings of $\mathbf{0}, \ldots, \mathbf{0}$. We will adapt standard techniques to sample the random coefficients of the linear combination.

The procedures corresponding to the above, $\pi_{\mathsf{Packed\text{-}Zero\text{-}DSS}}$, $\pi_{\mathsf{Check\text{-}Zero\text{-}DSS}}$, and $\pi_{\mathsf{Packed\text{-}DSS\text{-}Coins}}$ are presented in the full version.

4.3 Detectable Secret Sharing Protocol

Now, we are finally ready to present our $\Pi_{\mathsf{Packed\text{-}DSS}}$ protocol. For generating a sharing, a given dealer simply uses $\pi_{\mathsf{Packed\text{-}DSS\text{-}Share}}$ with degree $d_x \leftarrow t + \ell - 1$. For adding sharings and multiplying them by public vectors, the parties use $\pi_{\mathsf{Packed\text{-}DSS\text{-}Add}}$ and $\pi_{\mathsf{Packed\text{-}DSS\text{-}Mult}}$, respectively. The parties also keep track of when a given sharing is multiplied by a public vector. Then, for reconstruction, if the sharing is a linear combination of sharings that have not been multiplied by a public vector, the parties simply us $\pi_{\textsf{Packed-DSS-Rec}}$ to reconstruct it. Otherwise, the parties first re-randomize it by adding a random sharing $[\![\mathbf{0}]\!]_*$ to it, and then use $\pi_{\textsf{Packed-DSS-Rec}}$ to reconstruct it.

Protocol 6: $\Pi_{\textsf{Packed-DSS}}$

1. In the initialization phase, each party initializes instances of $\mathcal{F}_{\textsf{batch-IC}}$ with each other party as intermediary. The parties also run $\pi_{\textsf{Packed-Zero-DSS}}$ to compute a number N of random packed zero sharings $[\![\mathbf{0}_\tau]\!]_*$ for $\tau \in N$ (in parallel).
2. On input $(\textsf{share}, (\boldsymbol{s}_1, \dots, \boldsymbol{s}_m), \textsf{sid})$, P_i runs $\pi_{\textsf{Packed-DSS-Share}}(P_i, t + \ell - 1, \textsf{sid}, \boldsymbol{s}_1, \dots, \boldsymbol{s}_m)$ to share $[\![\boldsymbol{s}_{\textsf{sid}}]\!]$, then every party P_j sets $\textsf{isMult} \leftarrow 0$ and stores $(\textsf{sid}, (\textsf{isMult}, [\![\boldsymbol{s}_{\textsf{sid}}]\!]))$.
3. On input $(\textsf{reconstruct}, \textsf{sid})$, the parties first check $\textsf{isMult}$ stored with $\textsf{sid}$. If $\textsf{isMult} = 0$, then the parties run $\pi_{\textsf{Packed-DSS-Rec}}$ on $[\![\boldsymbol{s}_{\textsf{sid}}]\!]$ and output $\boldsymbol{s}_{\textsf{sid}}$. Otherwise, the parties (for next available $\tau \in [N]$), compute $[\![\boldsymbol{s}_{\textsf{sid}}]\!]_* + [\![\mathbf{0}_\tau]\!]_*$ and then run $\pi_{\textsf{Packed-DSS-Rec}}$ on it and output $\boldsymbol{s}_{\textsf{sid}}$.
4. On input $(\textsf{add}, \textsf{sid}_1, \textsf{sid}_2, \textsf{sid}_3)$, the parties compute $[\![\boldsymbol{s}_{\textsf{sid}_3}]\!] \leftarrow [\![\boldsymbol{s}_{\textsf{sid}_1}]\!] + [\![\boldsymbol{s}_{\textsf{sid}_2}]\!]$ (using $\pi_{\textsf{Packed-DSS-Add}}$). Then if $\textsf{isMult}_1 = \textsf{isMult}_2 = 0$, they set $\textsf{isMult}_3 \leftarrow 0$; otherwise, they set $\textsf{isMult}_3 \leftarrow 1$ Finally, they store $(\textsf{sid}_3, (\textsf{isMult}, [\![\boldsymbol{s}_{\textsf{sid}_3}]\!]))$.[a]
5. On input $(\textsf{mult}, (\boldsymbol{u}_1, \dots, \boldsymbol{u}_m), \textsf{sid}, \textsf{sid}')$: The parties first check that $\textsf{isMult}$ stored with $\textsf{sid}$ satisfies $\textsf{isMult} = 0$, and abort if not. If so, they compute $[\![\boldsymbol{s}_{\textsf{sid}'}]\!]_* \leftarrow [\![\boldsymbol{s}_{\textsf{sid}}]\!] * \boldsymbol{u}$ (using $\pi_{\textsf{Packed-DSS-Mult}}$). Finally, the parties store $(\textsf{sid}', (1, [\![\boldsymbol{s}_{\textsf{sid}'}]\!]_*))$.

[a] This also works if the sharings corresponding to $\textsf{sid}_1$ and/or $\textsf{sid}_2$ have higher degree $d_x = t + 2(\ell - 1)$; i.e., for sharings $[\![\boldsymbol{s}_{\textsf{sid}_1}]\!]_*$ and/or $[\![\boldsymbol{s}_{\textsf{sid}_2}]\!]_*$. In this case we store $[\![\boldsymbol{s}_{\textsf{sid}_3}]\!]_*$ of degree $d_x = t + 2(\ell - 1)$.

Efficiency of $\Pi_{\textsf{Packed-DSS}}$. Initialization uses $\pi_{\textsf{Packed-Zero-DSS}}$ in parallel, which takes $O(1)$ rounds. We will count the communication cost of generating each such zero sharing towards each such sharing that is reconstructed using it below.

Sharing uses $\pi_{\textsf{Packed-DSS-Share}}$, which costs $\textsf{P2P}(O(n^3 + n^2 m))$, $O(n^3) \times \textsf{BC}(O(1))$, and $O(n^2) \times \textsf{BC}(O(n + m))$. If $m, \ell = \Theta(n)$, then this is $\textsf{P2P}(O(n))$, $O(n) \times \textsf{BC}(O(1))$, and $O(1) \times \textsf{BC}(O(n))$ per underlying secret. It also takes $O(1)$ rounds.

Reconstruction possibly uses a zero sharing, generated from $\pi_{\textsf{Packed-Zero-DSS}}$, which costs $\textsf{P2P}(O(n^3 + n^2 m))$, $O(n^3) \times \textsf{BC}(O(1))$, and $O(n^2) \times \textsf{BC}(O(n + m))$ for this sharing. It then uses $\pi_{\textsf{Packed-DSS-Rec}}$, which costs $\textsf{P2P}(O(n^3 + n^2 m))$, $O(n^3) \times \textsf{BC}(O(1))$, and $O(n^2) \times \textsf{BC}(O(n + m))$. Altogether, this is $\textsf{P2P}(O(n^3 + n^2 m))$, $O(n^3) \times \textsf{BC}(O(1))$, and $O(n^2) \times \textsf{BC}(O(n + m))$. If $m, \ell = \Theta(n)$, then this is $\textsf{P2P}(O(n))$, $O(n) \times \textsf{BC}(O(1))$, and $O(1) \times \textsf{BC}(O(n))$ per underlying secret. It also takes $O(1)$ rounds.

Theorem 2. *$\Pi_{\textsf{Packed-DSS}}$ UC-realizes $\mathcal{F}_{\textsf{Packed-DSS}}$ in the $\mathcal{F}_{\textsf{batch-IC}}$-hybrid model for any $\ell \leq t/2$ and any $m = \texttt{poly}(\kappa)$, with probability $1 - \texttt{negl}(\kappa)$.*

The proof appears in the full version.

4.4 Extensions and Notation

The rest of the paper is devoted to using $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$, Functionality 4.1, to obtain honest majority MPC with G.O.D., with our claimed communication and round complexities. In the real world, this functionality corresponds to our packed DSS, but from now on we will work with the $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ abstraction, which allows us to ignoring details regarding shares, degrees, and other aspects only needed to instantiate this functionality. $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ is implicitly parameterized by ℓ and m, and it models a simple but quite powerful arithmetic black box: parties can store vectors of dimension $m\ell$, and these vectors can be computed on by adding them together, as well as multiplying them element-wise by public constant vectors. Furthermore, any stored vector can be reconstructed, and the only way for the adversary to stop it is to reveal the identities of at least $t - 2(\ell - 1)$ corrupt parties. Recall that, for $\boldsymbol{s} \in \mathbb{F}^{m\ell}$, we denote $[\![\boldsymbol{s}]\!]$ a value stored in $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ with $\mathsf{isMult} = 0$, and $[\![\boldsymbol{s}]\!]_*$ if $\mathsf{isMult} = 1$. Recall that given a public value $\boldsymbol{u} \in \mathbb{F}^{m\ell}$, it is possible to compute $[\![\boldsymbol{s}]\!]_* \leftarrow [\![\boldsymbol{s}]\!] * \boldsymbol{u}$. Addition of stored values and addition by public values is also possible. We use $[\![\boldsymbol{a}]\!] \leftarrow \mathsf{share}(\boldsymbol{a})$ to denote sharing, and $\boldsymbol{a} \leftarrow \mathsf{reconstruct}([\![\boldsymbol{a}]\!])$ to denote reconstruction (this also applies to $[\![\cdot]\!]_*$).

To be able to work with this functionality effectively, we will add to it a few helpful instructions that can be easily instantiated based on what we have seen so far. These include multiplication by scalars and addition by constants, which are particularly useful in the MPC context. The $[\![\cdot]\!]$ notation suggestively represents these operations. Finally, we add an instruction, whose call we abbreviate by $r \leftarrow \mathsf{rand}()$, which allows the honest parties to obtain a fresh random value. This can be instantiated with a communication of $O(n^4)$, *cf.* Sectionthe full version

5 Our MPC Protocol

We are now ready to put together the building blocks developed in previous sections to construct our final MPC protocol for honest majority with G.O.D.. As mentioned in technical overview (Sect. 1.3), the overall structure of our protocol is inspired on that of Turbopack [15], which is particularly suitable for the use of packed secret-sharing, a crucial tool we make use of in our work. While Turbopack uses plain packed secret-sharing, we make use of our optimized detectable secret-sharing, together with its reconstruction properties.

First, we define the MPC functionality with G.O.D. we aim at instantiating in this work. Let C be an arithmetic circuit over a finite field $\mathbb{F}$ comprised of inputs, addition and multiplication gates, and outputs. Each party P_i is responsible of providing a subset of the inputs. All parties are intended to receive the outputs. We use Greek letters α, β, γ, etc. to label wires in the circuit. We aim at instantiating $\mathcal{F}_{\mathsf{MPC}}$, Functionality 3, described below.

Functionality 3: $\mathcal{F}_{\mathsf{MPC}}$

The functionality proceeds as follows:

- **Receive inputs:** Upon receiving $(\mathsf{input}, P_i, x, \alpha)$ from an honest party P_i, or from the adversary if P_i is corrupt, where $x \in \mathbb{F}$ and α is an input wire assigned to P_i, store (α, x)
- **Compute the circuit:** Once all inputs have been provided, compute the circuit C on these inputs. For every output wire α, if its associated result is y, send (α, y) to all parties.

MPC for $t < n/3$. As part of our protocol, we will need an MPC protocol with G.O.D. for $t < n/3$. We model this with a functionality that behaves *almost* exactly the same as $\mathcal{F}_{\mathsf{MPC}}$, with the only difference being that (1) it interacts only with a subset of the parties, aborting if the subset has at least a 1/3 fraction of corruptions, and (2) it allows for *reactive* computation, meaning that different functions can be computed on the fly.[6] We denote this functionality by $\mathcal{F}_{\mathsf{MPC\text{-}t<n/3}}$ The recent work of [1] instantiates $\mathcal{F}_{\mathsf{MPC\text{-}t<n/3}}$ with linear communication $O(n|C'|)$ while maintaining the number of rounds $O(\mathsf{depth}(C'))$, where C' is the function being computed (we will use $\mathcal{F}_{\mathsf{MPC\text{-}t<n/3}}$ with a function C' that is slightly different to C, but has roughly the same size and depth). For the purpose of this section we use $[x]$ when a value $x \in \mathbb{F}$ has been provided as input to $\mathcal{F}_{\mathsf{MPC\text{-}t<n/3}}$, and we say "$P_i$ inputs x, obtaining $[x]$".

5.1 Offline Phase

We make use of two instances of $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$. To clearly differentiate between the two, we make the dependency of $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$ with ℓ and m explicit by writing $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}(\ell, m)$. The first instance $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}(\ell, m)$ allows parties to "share" or store vectors $\boldsymbol{s} \in \mathbb{F}^{m\cdot\ell}$, with $\ell = \lfloor \frac{n+6}{8} \rfloor$ and $m = n$. In what follows we use indistinctly "shared" and "stored" values/vectors, since even though we will be working in the $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}$-hybrid model, in the real world these corresponds to sharings. Recall from Sect. 4.4 that we use $[\![\boldsymbol{s}]\!]$ and $[\![\boldsymbol{s}]\!]_*$ to denote secret-shared vectors with $\mathsf{isMult} = 0$ and $\mathsf{isMult} = 1$. This, in the real world, corresponds to sharings of degree $t+(\ell-1)$ and $t+2(\ell-1)$ respectively, and the crucial difference is that first type of sharings allows for multiplications by public values whereas the latter does not. Reconstructions of $[\![\boldsymbol{s}]\!]$ and $[\![\boldsymbol{s}]\!]_*$ shared values may abort, at the expense of identifying more than $t-(\ell-1)$ or $t-2(\ell-1)$ corrupt parties respectively. The second instance $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}(1,1)$ shares individual values $s \in \mathbb{F}$, and these stored values are denoted by $\langle s \rangle$. Here, the adversary cannot cause abort when reconstructing shared values. Throughout this section, we denote $\mathbf{1} = (1, \ldots, 1) \in \mathbb{F}^{m\ell}$, and for $i \in [m\ell]$ we write $\mathbf{1}_i \in \mathbb{F}^{m\ell}$ for the vector of all zeros, except for the i-th entry, which equals 1.

[6] $\mathcal{F}_{\mathsf{MPC}}$, as defined, is not reactive. However, this is only for presentation and it is not hard to extend our protocol to support reactive computation.

Our preprocessing is as in Turbopack [15]. First, we group multiplication gates in each layer in groups of $m \cdot \ell$ gates each, and we do the same with the input wires associated to each party, as well as the output wires. Each circuit wire α that is not the output of an addition gate has associated to it a random mask $\lambda_\alpha \in \mathbb{F}$. If two wires α, β are added to obtain wire γ, then $\lambda_\gamma := \lambda_\alpha + \lambda_\beta$. The preprocessing consists of sharings $[\![\boldsymbol{\lambda_\alpha}]\!]_*$ for every output group $\boldsymbol{\alpha}$, and sharings $([\![\boldsymbol{\lambda_\alpha}]\!], [\![\boldsymbol{\lambda_\beta}]\!], [\![\boldsymbol{\lambda_\alpha} \star \boldsymbol{\lambda_\beta} - \boldsymbol{\lambda_\gamma}]\!]_*)$ for every group of multiplication gates with inputs $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and outputs $\boldsymbol{\gamma}$. In addition, every party P_i having an input wire α must learn λ_α. For the case of a restart, we also require every such λ_α for input wires to be VSS'ed as $\langle \lambda_\alpha \rangle$.[7] This is captured by $\mathcal{F}_{\mathsf{Prep}}$, Functionality 4.

Functionality 4: $\mathcal{F}_{\mathsf{Prep}}$

Extension of the Packed DSS functionality. $\mathcal{F}_{\mathsf{Prep}}$ has all of the instructions of $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}(\ell, m)$ and $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}(1, 1)$ (extended as in Section 4.4).[a]

Sample random masks. Sample the following values

1. For each circuit wire α that is *not* the result of an addition gate, sample a random $\lambda_\alpha \in \mathbb{F}$ and store (α, λ_α).
2. For every addition gate with inputs α, β and output γ, compute $\lambda_\gamma = \lambda_\alpha + \lambda_\beta$ and store (γ, λ_γ)

Input and output sharings. For every group of $m\ell$ input gates with labels $\boldsymbol{\alpha}$ belonging to party P_i:

1. Send $\boldsymbol{\lambda_\alpha}$ to P_i,
2. Store $\langle \lambda_{\alpha_j} \rangle$ for $j \in [m\ell]$.

For every group of $m\ell$ output gates with labels $\boldsymbol{\alpha}$, store $[\![\boldsymbol{\lambda_\alpha}]\!]$

Multiplication gates. For every group of $m\ell$ multiplication gates with left input labels $\boldsymbol{\alpha}$, right input labels $\boldsymbol{\beta}$, and output labels $\boldsymbol{\gamma}$, the functionality stores $([\![\boldsymbol{\lambda_\alpha}]\!], [\![\boldsymbol{\lambda_\beta}]\!], [\![\boldsymbol{\lambda_\alpha} \star \boldsymbol{\lambda_\beta} - \boldsymbol{\lambda_\gamma}]\!]_*)$.

[a] Values stored as in $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}(\ell, m)$ are kept in a separate dictionary than these from $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}(1, 1)$.

Multiplication Triple Generation. For our preprocessing we will require uniformly random multiplication triples $([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!], [\![\boldsymbol{c}]\!]_*)$, with $\boldsymbol{c} = \boldsymbol{a} \star \boldsymbol{b}$. To this end, we show how to extend the techniques from [10], which are set in the standard (non-packed) secret-sharing setting, to the packed secret-sharing regime we use in our work. We choose the techniques from [10] since, in contrast to other approaches such as [13], no "degree-$2t$ computations" are needed, and instead all shares are either degree $t + (\ell - 1)$, or $t + 2(\ell - 1)$. This is crucial for us, where we require reconstruction to either succeed, or identify a large set of corrupt parties.

[7] Having each input to be VSS'ed adds an extra factor of n with respect to the number of inputs. We present in the full version a variant that is more suitable incase there are many more inputs than outputs.

It is not difficult to adapt the techniques from [10] to our setting, and we discuss this in the full version. For the purpose of this section, we simply mention that there is a procedure $\pi_{\textsf{triple-generation}}$ (Procedure ?? in the full version) that generates a single batched triple $([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!], [\![\boldsymbol{c}]\!]_*)$ which will be uniform and independent from the view of corrupt parties, as captured by the following Lemma (proven in the full version).

Lemma 5. $\pi_{\textsf{triple-generation}}$ *outputs a batched triple* $([\![\boldsymbol{a}_{new}]\!], [\![\boldsymbol{b}_{new}]\!], [\![\boldsymbol{c}_{new}]\!]_*)$ *which is uniform and independent from the view of an adversary corrupting at most t parties except with* $\texttt{negl}(\kappa)$ *failure probability.*

The overall cost of $\pi_{\textsf{triple-generation}}$ is $\textsf{P2P}(O(n^4))$, $O(n^4) \times \textsf{BC}(1)$. Since $\pi_{\textsf{triple-generation}}$ outputs a single batched triple containing $O(m\ell) = O(n^2)$ triples, the amortized communication cost per triple generation is $\textsf{P2P}(O(n^2))$, $O(n^2) \times \textsf{BC}(1)$.

Useful Procedures. Before we describe the protocol that instantiates $\mathcal{F}_{\textsf{Prep}}$, we describe a few useful procedures. The first, $\pi_{\textsf{Input-Sharings}}(P_i)$ (Procedure ??), enables the parties to obtain random sharings of the form $([\![r \cdot \mathbf{1}]\!], \langle r \rangle)$, where P_i knows r. This will be important for providing inputs, with the VSS part enabling restarting without input modification. The procedure follows along the same lines as $\pi_{\textsf{Check-Zero-DSS}}$, Procedure ??, which lets P_i distribute these sharings and the parties check them via random linear combinations. The second procedure, which we denote by $\pi_{\textsf{Rand-Sharings}}$ (Procedure ??), allows the parties to obtain $[\![r \cdot \mathbf{1}]\!]$, where $r \in \mathbb{F}$ is uniformly random and unknown to any party. This first uses ideas as in the first procedure to let each party distribute one such sharing correctly, and then, similarly to $\pi_{\textsf{Packed-Zero-DSS}}$ (Procedure ??), we can use standard techniques based on Vandermonde matrices to extract uniformly random sharings. The full descriptions of the procedures appear in the full version. The following two Lemmas are proven similarly to Lemma ?? in the full version, we omit their proof.

Lemma 6. *Except with probability* $\texttt{negl}(\kappa)$*, the output* $[\![r \cdot \mathbf{1}]\!], \langle r \rangle$ *produced by* $\pi_{\textsf{Input-Sharings}}(P_i)$ *is correct, and for an honest P_i, the secret r is distributed randomly given the corrupted parties' shares.*

Lemma 7. *Except with probability* $\texttt{negl}(\kappa)$*, the outputs* $([\![s_1 \cdot \mathbf{1}]\!], \dots, [\![s_{n-t} \cdot \mathbf{1}]\!])$ *produced by* $\pi_{\textsf{Rand-Sharings}}$ *are correct, and are distributed randomly given the corrupted parties' shares.*

Preprocessing Protocol. We are finally ready to present our protocol for instantiating $\mathcal{F}_{\textsf{Prep}}$. This is given in $\Pi_{\textsf{Prep}}$, Protocol 7 below.

Protocol 7: Π_{Prep}

The protocol makes use of two functionalities $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}(\ell, m)$ and $\mathcal{F}_{\mathsf{Packed\text{-}DSS}}(1, 1)$, and every command regarding packed DSS is forwarded to these functionalities. For the other commands:

Input groups. For every input wire α associated to party P_i, call $([\![\lambda_\alpha \cdot \mathbf{1}]\!], \langle\lambda_\alpha\rangle) \leftarrow \pi_{\mathsf{Input\text{-}Sharings}}(P_i)$ for $i \in [n]$.[a]

Sampling random masks. For every wire α that is either an output of a multiplication gate, or an input wire, the parties call $[\![\lambda_\alpha \cdot \mathbf{1}]\!] \leftarrow \pi_{\mathsf{Rand\text{-}Sharings}}()$. After this note that, locally, they can compute $[\![\lambda_\gamma \cdot \mathbf{1}]\!]$ for every wire γ that is the output of an addition gate by adding the corresponding shares. This means they have $[\![\lambda_{\alpha_i} \cdot \mathbf{1}]\!]$ for *every* circuit wire α.

Output groups. For an output group $\boldsymbol{\alpha}$, the parties take the sharings $[\![\lambda_{\alpha_i} \cdot \mathbf{1}]\!]$ for $i \in [m\ell]$ from the previous step and output $[\![\boldsymbol{\lambda}_\alpha]\!]_* = \sum_{i=1}^{m\ell} \mathbf{1}_i \cdot [\![\lambda_{\alpha_i} \mathbf{1}]\!]$.

Multiplication groups. For a multiplication group with input wires $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and output wires $\boldsymbol{\gamma}$, the parties do the following:

1. Call $\pi_{\mathsf{triple\text{-}generation}}$ to obtain $([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!], [\![\boldsymbol{a} \star \boldsymbol{b}]\!]_*)$
2. The parties proceed as before, obtaining $[\![\boldsymbol{\lambda}_\alpha]\!]_*$ and $[\![\boldsymbol{\lambda}_\beta]\!]_*$. Similarly, they get $[\![\boldsymbol{\lambda}_\gamma]\!]_*$
3. Locally compute $[\![\boldsymbol{d}]\!]_* \leftarrow [\![\boldsymbol{\lambda}_\alpha]\!]_* - [\![\boldsymbol{a}]\!]$ and $[\![\boldsymbol{e}]\!]_* \leftarrow [\![\boldsymbol{\lambda}_\beta]\!]_* - [\![\boldsymbol{b}]\!]$
4. Call $\boldsymbol{d} \leftarrow \mathsf{reconstruct}([\![\boldsymbol{d}]\!]_*)$ and $\boldsymbol{e} \leftarrow \mathsf{reconstruct}([\![\boldsymbol{e}]\!]_*)$
5. Locally compute $[\![\boldsymbol{\lambda}_\alpha]\!] \leftarrow \boldsymbol{d} + [\![\boldsymbol{a}]\!]$, $[\![\boldsymbol{\lambda}_\beta]\!] \leftarrow \boldsymbol{e} + [\![\boldsymbol{b}]\!]$, and

$$[\![\boldsymbol{\lambda}_\alpha \star \boldsymbol{\lambda}_\beta - \boldsymbol{\lambda}_\gamma]\!]_* \leftarrow \boldsymbol{d} \cdot [\![\boldsymbol{a}]\!] + \boldsymbol{e} \cdot [\![\boldsymbol{b}]\!] + \boldsymbol{d} \star \boldsymbol{e} + [\![\boldsymbol{a} \star \boldsymbol{b}]\!]_* - [\![\boldsymbol{\lambda}_\gamma]\!]_*,$$

   and output $([\![\boldsymbol{\lambda}_\alpha]\!], [\![\boldsymbol{\lambda}_\beta]\!], [\![\boldsymbol{\lambda}_\alpha \star \boldsymbol{\lambda}_\beta - \boldsymbol{\lambda}_\gamma]\!]_*)$

Abort. Note that, if any of the steps above results in abort, then a set T of corrupt parties with $|T| > t - 2(\ell - 1)$ is identified. In this case the parties output this set.

[a] If the parties abort during this step, P_i's inputs will be disregarded, as we know they are corrupted.

Theorem 3. *Π_{Prep} UC-realizes $\mathcal{F}_{\mathsf{Prep}}$ in the $(\mathcal{F}_{\mathsf{Packed\text{-}DSS}}(\ell, m), \mathcal{F}_{\mathsf{Packed\text{-}DSS}}(1, 1))$-hybrid model, with probability $1 - \mathtt{negl}(\kappa)$.*

The proof appears in the full version.

Communication Complexity. We now calculate the communication cost of Π_{Prep} by calculating the cost of different parts:

1. Input groups: This step invokes $\pi_{\mathsf{Input\text{-}Sharings}}$ k times where k is the number of input wires. Each invocation of $\pi_{\mathsf{Input\text{-}Sharings}}$ costs $\mathsf{P2P}(O(n^3))$, $O(n^3) \times \mathsf{BC}(1)$ (assuming the cost of $\mathsf{rand}()$ is amortized across n parties). Therefore, the total cost of this step is $\mathsf{P2P}(O(|C|n^3))$, $O(|C|n^3) \times \mathsf{BC}(1)$.
2. Sampling random masks: This step invokes $\pi_{\mathsf{Rand\text{-}Sharings}}$ $O(|C|/n)$ times. Each invocation of $\pi_{\mathsf{Rand\text{-}Sharings}}$ costs $\mathsf{P2P}(O(n^4))$, $O(n^4) \times \mathsf{BC}(1)$. (assuming the cost of $\mathsf{rand}()$ is amortized across n parties). Therefore, the total cost of this step is $\mathsf{P2P}(O(|C|n^3))$, $O(|C|n^3) \times \mathsf{BC}(1)$.

3. Multiplication groups: Let $k = |C|/n^2$ be the number of multiplication groups. This step invokes $\pi_{\mathsf{triple\text{-}generation}}$ k times where each invocation costs $\mathsf{P2P}(O(n^4))$, $O(n^4) \times \mathsf{BC}(1)$. Also, it performs a beaver multiplication (same as π_{Beaver}presented in the full version) k times where each multiplication costs $\mathsf{P2P}(O(n^3))$, $O(n^3) \times \mathsf{BC}(1)$. Therefore, the total cost of this step is $\mathsf{P2P}(O(|C|n^2))$, $O(|C|n^2) \times \mathsf{BC}(1)$.

Summing up all the above costs, the overall communication cost of Π_{Prep} is $\mathsf{P2P}(O(|C|n^3))$, $O(|C|n^3) \times \mathsf{BC}(1)$.

Remark 2. (On function-dependent/independent preprocessing.). As in [15], we can easily make our offline phase function-independent without affecting our asymptotic communication in the online phase. For this, the offline phase consists only of generating sharings of the form $[\![r \cdot \mathbf{1}]\!]$ and $([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!], [\![\boldsymbol{a} \star \boldsymbol{b}]\!]_*)$ (which is function-independent), and the part of Π_{Prep} that turns these into the function-dependent $([\![\boldsymbol{\lambda_\alpha}]\!], [\![\boldsymbol{\lambda_\beta}]\!], [\![\boldsymbol{\lambda_\alpha} \star \boldsymbol{\lambda_\beta} - \boldsymbol{\lambda_\gamma}]\!]_*)$ is moved to the online phase Π_{MPC}. Crucially, the communication complexity of these steps is $O(n|C|)$.

5.2 Online Phase

Finally, we present Π_{MPC}, Protocol 8, which instantiates $\mathcal{F}_{\mathsf{MPC}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model. This corresponds to the online phase, and at a high level it proceeds by maitaining the following invariant. For a wire α and a given assignment to the circuit inputs, let us denote by v_α the value held by wire α. The protocol maintains that, for every wire α, the parties have the values $\mu_\alpha := v_\alpha - \lambda_\alpha$ in the clear. This is ensured all the way up to the outputs, point in which the parties can reconstruct the associated masks and obtain the outputs. A major difference with respect to Turbopack [15] is that, in our case, we need to handle the case in which any of the steps that involve reconstructions—either in the offline or online phase—result in abort. For this, we let the parties restart the computation, kicking out the identified corrupt parties, which guarantees the new corruption threshold is 1/3. The parties make use of the $t < n/3$ MPC functionality for this $\mathcal{F}_{\mathsf{MPC\text{-}t<n/3}}$, but before doing that they use the initial VSS'ed masks $\langle\lambda_\alpha\rangle$ to ensure that the inputs provided to $\mathcal{F}_{\mathsf{MPC\text{-}t<n/3}}$ are consistent with these from the initial execution that resulted in abort.

Protocol 8: Π_{MPC}

This protocol makes use of $\mathcal{F}_{\mathsf{Prep}}$ and $\mathcal{F}_{\mathsf{MPC\text{-}t<n/3}}$.

Preprocessing. The parties call $\mathcal{F}_{\mathsf{Prep}}$ to obtain:
- $[\![\boldsymbol{\lambda_\alpha}]\!]_*$ for every output group $\boldsymbol{\alpha}$
- $([\![\boldsymbol{\lambda_\alpha}]\!], [\![\boldsymbol{\lambda_\beta}]\!], [\![\boldsymbol{\lambda_\alpha} \star \boldsymbol{\lambda_\beta} - \boldsymbol{\lambda_\gamma}]\!]_*)$ for every multiplication group with inputs $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and outputs $\boldsymbol{\gamma}$.
- For every input group $\boldsymbol{\alpha}$ assigned to a party P_i, this party knows $\boldsymbol{\lambda_\alpha}$, and the parties have $\langle\lambda_{\alpha_1}\rangle, \ldots, \langle\lambda_{\alpha_{m\ell}}\rangle$

Input Gates. For a group of input gates $\boldsymbol{\alpha}$ owned by a party P_i, this party, who knows $\boldsymbol{\lambda}_{\boldsymbol{\alpha}}$ from the preprocessing, and also knows its input $\boldsymbol{v}_{\boldsymbol{\alpha}}$, broadcasts $\boldsymbol{\mu}_{\boldsymbol{\alpha}} = \boldsymbol{v}_{\boldsymbol{\alpha}} - \boldsymbol{\lambda}_{\boldsymbol{\alpha}}$.

Addition Gates. For a group of addition gates with inputs $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and outputs $\boldsymbol{\gamma}$, the parties locally add $\boldsymbol{\mu}_{\boldsymbol{\gamma}} \leftarrow \boldsymbol{\mu}_{\boldsymbol{\alpha}} + \boldsymbol{\mu}_{\boldsymbol{\beta}}$.

Multiplication Gates. For a group of multiplication gates with inputs $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and outputs $\boldsymbol{\gamma}$, the parties proceed as follows:

1. Locally compute

$$[\![\boldsymbol{\mu}_{\boldsymbol{\gamma}}]\!]_* \leftarrow \boldsymbol{\mu}_{\boldsymbol{\alpha}} \cdot [\![\boldsymbol{\lambda}_{\boldsymbol{\beta}}]\!] + \boldsymbol{\mu}_{\boldsymbol{\beta}} \cdot [\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!] + \boldsymbol{\mu}_{\boldsymbol{\alpha}} \star \boldsymbol{\mu}_{\boldsymbol{\beta}} + [\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}} \star \boldsymbol{\lambda}_{\boldsymbol{\beta}} - \boldsymbol{\lambda}_{\boldsymbol{\gamma}}]\!]_*$$

2. Call $\boldsymbol{\mu}_{\boldsymbol{\gamma}} \leftarrow \mathsf{reconstruct}([\![\boldsymbol{\mu}_{\boldsymbol{\gamma}}]\!]_*)$.

Output Gates. Given a group of output wires $\boldsymbol{\alpha}$, call $\boldsymbol{\lambda}_{\boldsymbol{\alpha}} \leftarrow \mathsf{reconstruct}([\![\boldsymbol{\lambda}_{\boldsymbol{\alpha}}]\!]_*)$, and return the output $\boldsymbol{v}_{\boldsymbol{\alpha}} = \boldsymbol{\lambda}_{\boldsymbol{\alpha}} + \boldsymbol{\mu}_{\boldsymbol{\alpha}}$.

Abort and restart. If any of the calls above results in abort, a set T of corrupt parties with $|T| > t - 2(\ell - 1)$ is identified. The new set of parties is $\{P_1, \ldots, P_n\} \setminus T$, where $n' = n - |T|$ and $t' = t - |T|$, and they execute the following.

- For every input wire α that belongs to $P_i \in T$, the parties call $\lambda_\alpha \leftarrow \mathsf{reconstruct}(\langle \lambda_\alpha \rangle)$ and set $v_\alpha = \mu_\alpha + \lambda_\alpha$.
- For every $P_i \notin T$, let $\alpha_1, \ldots, \alpha_M$ be the input wires that belongs to P_i. The parties do the following:
  1. P_i inputs λ_{α_j} in $\mathcal{F}_{\mathsf{MPC\text{-}t<n/3}}$, obtaining $[\lambda_{\alpha_j}]$, for $j \in [M]$.
  2. P_i samples $r \leftarrow_\$ \mathbb{F}$ and calls $\langle r \rangle \leftarrow \mathsf{share}(r)$.[a] P_i also inputs r in $\mathcal{F}_{\mathsf{MPC\text{-}t<n/3}}$, obtaining $[r]$.
  3. Parties call $c_1, \ldots, c_M \leftarrow \mathsf{ttrand}()$
  4. Parties compute $\langle z \rangle \leftarrow \langle r \rangle + \sum_{j=1}^{M} c_j \cdot \langle \lambda_{\alpha_j} \rangle$ and call $z \leftarrow \mathsf{reconstruct}(\langle z \rangle)$
  5. Use $\mathcal{F}_{\mathsf{MPC\text{-}t<n/3}}$ to compute the following function.
     - The inputs are $[\lambda_{\alpha_1}], \ldots, [\lambda_{\alpha_M}], [r]$ as above
     - The function first computes $[z'] \leftarrow [r] + \sum_{j=1}^{M} c_j \cdot [\lambda_{\alpha_j}]$, and outputs 1 if $z' = z$, and 0 otherwise.
  6. If the output is 0, the parties call $\lambda_{\alpha_j} \leftarrow \mathsf{reconstruct}(\langle \lambda_{\alpha_j} \rangle)$ and set $v_{\alpha_j} = \mu_{\alpha_j} + \lambda_{\alpha_j}$, for $j \in [M]$. The party P_i is added to T.
- The parties then use $\mathcal{F}_{\mathsf{MPC\text{-}t<n/3}}$ to compute the following function and return its outputs:
  - The (secret) inputs are, for each $P_i \notin T$, $[\lambda_{\alpha_1}], \ldots, [\lambda_{\alpha_M}]$ as above.
  - For every such values, the function first computes $[v_{\alpha_j}] = \mu_{\alpha_j} + [\lambda_{\alpha_j}]$ (recall that μ_{α_j} is public, as it is broadcast in the input phase).
  - Using these inputs, together with the public inputs v_{α_j} for $P_i \in T$, compute the circuit C. These outputs are the outputs of the function.

[a] If this sharing aborts, the parties skip to step 6., since P_i must be corrupted.

We prove the following in the full version.

Theorem 4. *Π_{MPC} UC-realizes $\mathcal{F}_{\mathsf{MPC}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model, with probability $1 - \mathtt{negl}(\kappa)$.*

Communication complexity. We now calculate the communication cost of Π_{MPC} by calculating the cost of different parts:

1. Input gates: This involves each party broadcasting a batch of inputs per input group that it owns. Across all parties and all input groups possible, the cost of this step is bounded by $\mathsf{P2P}(O(|C|n + n^4))$, $O(|C|n + n^4) \times \mathsf{BC}(1)$.
2. Addition gates: This step is local so there is no communication cost.
3. Multiplication gates: Let $k = |C|/n^2$ be the total number of groups of multiplication gates in the circuit. For each group, we invoke a single $\mathsf{reconstruct}$ which requires $\mathsf{P2P}(O(n^3))$, $O(n^3) \times \mathsf{BC}(1)$. Hence, the overall cost of this step is $\mathsf{P2P}(O(|C|n))$, $O(|C|n) \times \mathsf{BC}(1)$.
4. Output gates: Let $k = |C|/n$ be the total number of groups of output gates in the circuit. For each group, we invoke a single $\mathsf{reconstruct}$ which requires $\mathsf{P2P}(O(n^3))$, $O(n^3) \times \mathsf{BC}(1)$. Hence, the overall cost of this step is $\mathsf{P2P}(O(|C|n))$, $O(|C|n) \times \mathsf{BC}(1)$.
5. Abort and restart: Let c_I be the number of input wires. The cost of this step is $\mathsf{P2P}(O(|C|n + c_I n^3))$, $O(c_I n^3) \times \mathsf{BC}(1)$.

Combining all the costs, we get that the overall communication cost of Π_{MPC} in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model is $\mathsf{P2P}(O(|C|n + c_I n^3 + n^4))$, $O(|C|n + c_I n^3 + n^4) \times \mathsf{BC}(1)$. Assuming $C >> c_I \cdot n^2$, we get communication cost of $\mathsf{P2P}(O(|C|n+n^4))$, $O(|C|n + n^4) \times \mathsf{BC}(1)$.

Acknowledgments. This paper was prepared in part for information purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful. 2024 JP Morgan Chase & Co. All rights reserved.

References

1. Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. "Detect, Pack and Batch: Perfectly-Secure MPC with Linear Communication and Constant Expected Time". In: *Advances in Cryptology – EUROCRYPT 2023, Part II*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14005. Lecture Notes in Computer Science. Lyon, France: Springer, Heidelberg, Germany, 2023, pp. 251–281. DOI: https://doi.org/10.1007/978-3-031-30617-4_9.
2. Ittai Abraham, Gilad Asharov, and Avishay Yanai. "Efficient Perfectly Secure Computation with Optimal Resilience". In: *TCC 2021: 19th Theory of Cryptography Conference, Part II*. Ed. by Kobbi Nissim and Brent Waters. Vol. 13043. Lecture Notes in Computer Science. Raleigh, NC, USA: Springer, Heidelberg, Germany, 2021, pp. 66–96. DOI: https://doi.org/10.1007/978-3-030-90453-1_3.
3. Benny Applebaum, Eliran Kachlon, and Arpita Patra. "The Round Complexity of Statistical MPC with Optimal Resiliency". In: *Cryptology ePrint Archive* (2023).
4. Donald Beaver. "Efficient Multiparty Protocols Using Circuit Randomization". In: *Advances in Cryptology - CRYPTO'91*. Ed. by Joan Feigenbaum. Vol. 576. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 1992, pp. 420–432. DOI: https://doi.org/10.1007/3-540-46766-1_34.
5. Zuzana Beerliová-Trubíniová and Martin Hirt. "Efficient Multi-party Computation with Dispute Control". In: *TCC 2006: 3rd Theory of Cryptography Conference*. Ed. by Shai Halevi and Tal Rabin. Vol. 3876. Lecture Notes in Computer Science. New York, NY, USA: Springer, Heidelberg, Germany, 2006, pp. 305–328. DOI: https://doi.org/10.1007/11681878_16.
6. Zuzana Beerliová-Trubíniová and Martin Hirt. "Perfectly-Secure MPC with Linear Communication Complexity". In: *TCC 2008: 5th Theory of Cryptography Conference*. Ed. by Ran Canetti. Vol. 4948. Lecture Notes in Computer Science. San Francisco, CA, USA: Springer, Heidelberg, Germany, 2008, pp. 213–230. DOI: https://doi.org/10.1007/978-3-540-78524-8_13.
7. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)". In: *20th Annual ACM Symposium on Theory of Computing*. Chicago, IL, USA: ACM Press, 1988, pp. 1–10. DOI: https://doi.org/10.1145/62212.62213.
8. Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. "Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority". In: *Advances in Cryptology - CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2012, pp. 663–680. DOI: https://doi.org/10.1007/978-3-642-32009-5_39.
9. David Chaum, Claude Crépeau, and Ivan Damgård. "Multiparty Unconditionally Secure Protocols (Extended Abstract)". In: *20th Annual ACM Symposium on Theory of Computing*. Chicago, IL, USA: ACM Press, 1988, pp. 11–19. DOI: https://doi.org/10.1145/62212.62214.
10. Ashish Choudhury and Arpita Patra. "An Efficient Framework for Unconditionally Secure Multiparty Computation". In: *IEEE Transactions on Information Theory* 63.1 (2017), pp. 428–468. DOI: https://doi.org/10.1109/TIT.2016.2614685.
11. Ronald Cramer, Ivan Damgård, Stefan Dziembowski, Martin Hirt, and Tal Rabin. "Efficient Multiparty Computations Secure Against an Adaptive Adversary". In: *Advances in Cryptology - EUROCRYPT'99*. Ed. by Jacques Stern. Vol. 1592. Lecture Notes in Computer Science. Prague, Czech Republic: Springer, Heidelberg, Germany, 1999, pp. 311–326. DOI: https://doi.org/10.1007/3-540-48910-X_22.

12. Ivan Damgård, Kasper Green Larsen, and Jesper Buus Nielsen. "Communication Lower Bounds for Statistically Secure MPC, With or Without Preprocessing". In: *Advances in Cryptology - CRYPTO 2019, Part II*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11693. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2019, pp. 61–84. DOI: https://doi.org/10.1007/978-3-030-26951-7_3.
13. Ivan Damgård and Jesper Buus Nielsen. "Scalable and Unconditionally Secure Multiparty Computation". In: *Advances in Cryptology - CRYPTO 2007*. Ed. by Alfred Menezes. Vol. 4622. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2007, pp. 572–590. DOI: https://doi.org/10.1007/978-3-540-74143-5_32.
14. Daniel Escudero and Serge Fehr. "On Fully-Secure Honest Majority MPC Without n2 Round Overhead". In: *Progress in Cryptology - LATINCRYPT 2021: 7th International Conference on Cryptology and Information Security in Latin America. Ed.* by Patrick Longa and Carla Ràfols. Vol. 12912. Lecture Notes in Computer Science. Bogotá, Colombia: Springer, Heidelberg, Germany, 2021, pp. 47–66. DOI: https://doi.org/10.1007/978-3-031-44469-2_3.
15. Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. "TurboPack: Honest Majority MPC with Constant Online Communication". In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. Los Angeles, CA, USA: ACM Press, 2022, pp. 951–964. DOI: https://doi.org/10.1145/3548606.3560633.
16. Matthew K. Franklin and Moti Yung. "Communication Complexity of Secure Computation (Extended Abstract)". In: *24th Annual ACM Symposium on Theory of Computing*. Victoria, BC, Canada: ACM Press, 1992, pp. 699–710. DOI: https://doi.org/10.1145/129712.129780.
17. Vipul Goyal, Yanyi Liu, and Yifan Song. "Communication-Efficient Unconditional MPC with Guaranteed Output Delivery". In: *Advances in Cryptology - CRYPTO 2019, Part II*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11693. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2019, pp. 85–114. DOI: https://doi.org/10.1007/978-3-030-26951-7_4.
18. Vipul Goyal, Yifan Song, and Chenzhi Zhu. "Guaranteed Output Delivery Comes Free in Honest Majority MPC". In: *Advances in Cryptology - CRYPTO 2020, Part II*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12171. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2020, pp. 618–646. DOI: https://doi.org/10.1007/978-3-030-56880-1_22.
19. Martin Hirt, Ueli M. Maurer, and Bartosz Przydatek. "Efficient Secure Multiparty Computation". In: *Advances in Cryptology - ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Vol. 1976. Lecture Notes in Computer Science. Kyoto, Japan: Springer, Heidelberg, Germany, 2000, pp. 143–161. DOI: https://doi.org/10.1007/3-540-44448-3_12.
20. Yuval Ishai and Eyal Kushilevitz. "Perfect constant-round secure computation via perfect randomizing polynomials". In: *Automata, Languages and Programming: 29th International Colloquium, ICALP 2002 Málaga, Spain, July 8-13, 2002 Proceedings 29*. Springer. 2002, pp. 244–256.
21. Yuval Ishai and Eyal Kushilevitz. "Randomizing polynomials: A new representation with applications to round-efficient secure computation". In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE. 2000, pp. 294–304.

22. Yuval Ishai, Eyal Kushilevitz, Manoj Prabhakaran, Amit Sahai, and Ching- Hua Yu. "Secure Protocol Transformations". In: *Advances in Cryptology - CRYPTO 2016, Part II*. Ed. by Matthew Robshaw and Jonathan Katz. Vol. 9815. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, 2016, pp. 430–458. DOI: https://doi.org/10.1007/978-3-662-53008-5_15.
23. Arpita Patra and C. Pandu Rangan. *Communication and Round Efficient Information Checking Protocol*. 2010. arXiv: 1004.3504 [cs.CR].
24. Tal Rabin and Michael Ben-Or. "Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract)". In: *21st Annual ACM Symposium on Theory of Computing*. Seattle, WA, USA: ACM Press, 1989, pp. 73–85. DOI: https://doi.org/10.1145/73007.73014.

Direct FSS Constructions for Branching Programs and More from PRGs with Encoded-Output Homomorphism

Elette Boyle[1,2](✉), Lisa Kohl[3], Zhe Li[3], and Peter Scholl[4]

[1] Reichman University, Herzliya, Israel
[2] NTT Research, Sunnyvale, USA
eboyle@alum.mit.edu
[3] Cryptology Group, CWI Amsterdam, Amsterdam, The Netherlands
lisa.kohl@cwi.nl, lizh0048@e.ntu.edu.sg
[4] Aarhus University, Aarhus, Denmark
peter.scholl@cs.au.dk

Abstract. Function secret sharing (FSS) for a class $\mathcal{F}$ allows to split a secret function $f \in \mathcal{F}$ into (succinct) secret shares f_0, f_1, such that for all $x \in \{0,1\}^n$ it holds $f_0(x) - f_1(x) = f(x)$. FSS has numerous applications, including private database queries, nearest neighbour search, private heavy hitters and secure computation in the preprocessing model, where the supported class $\mathcal{F}$ translates to richness in the application. Unfortunately, concretely efficient FSS constructions are only known for very limited function classes.

In this work we introduce the notion of pseudorandom generators with encoded-output homomorphism (EOH-PRGs), and give direct FSS constructions for branching programs and more based on this primitive. Further, we give constructions of FSS for deterministic finite automatas (DFAs) from a KDM secure variant of EOH-PRGs.

– *New abstractions.* Following the work of Alamati et al. (EUROCRYPT '19), who classify minicrypt primitives with algebraic structure and their applications, we capture the essence of our FSS constructions in the notion of EOH-PRG, paving the road towards future efficiency improvements via new instantiations of this primitive. The abstraction of EOH-PRG and its instantiations may be of independent interest, as it is an approximate substitution of an ideal homomorphic PRG.
– *Better efficiency.* We show that EOH-PRGs can be instantiated from LWE and a small-exponent variant of the DCR assumption. A theoretical analysis of our instantiations suggest efficiency improvements over the state of the art both in terms of key size and evaluation time: We show that our FSS instantiations lead to smaller key sizes, improving over previous constructions by a factor of 3.5 and more. For branching programs our FSS constructions show considerably improved run time by avoiding the expensive generic transformation via universal circuits, shaving off a factor of w and more in the number of abstract operations, where w corresponds to an upper bound on the width of the underlying class of branching programs.

© International Association for Cryptologic Research 2025
K.-M. Chung and Y. Sasaki (Eds.): ASIACRYPT 2024, LNCS 15489, pp. 266–298, 2025.
https://doi.org/10.1007/978-981-96-0938-3_9

– *New feasibility.* We show that our instantiations of EOH-PRGs additionally support a form of KDM-security, without requiring an additional circular-security assumption. Based on this, we give the first FSS construction for DFAs which supports the evaluation of inputs of a-priori unbounded length without relying on FHE.
– *Applications.* We outline applications of our FSS constructions including pattern matching with wild cards, image matching, nearest neighbor search and regular expression matching.

1 Introduction

Boyle, Gilboa and Ishai [17] introduced the notion of *function secret sharing* in 2015. Function secret sharing for a class of functions $\mathcal{F}$ allows to split up a function $f\colon \{0,1\}^n \to \mathbb{G}$ from $\mathcal{F}$ into secret shares f_0, f_1, such that for all $x \in \{0,1\}^n$ it holds $f_0(x) - f_1(x) = f(x)$. If $f\colon \{0,1\}^n \to \mathbb{G}$ is an arbitrary function, its description size can in general scale with 2^n, and thus there is no hope to get compact secret shares. On the other hand, if f is from a class of functions with *succinct* description, one can hope to split the function up into *succinct* secret shares. As shown in [17], when relaxing the secrecy condition to computational (i.e., requiring that no computationally bounded adversary holding only a subset of the shares can derive information about the function within the function class), this can indeed be achieved.

Function secret sharing schemes have been used in numerous applications, such as multi-server private-information retrieval [17,33], oblivious RAM [30], anonymous broadcast messaging [25], private database queries [45], nearest neighbour search [42], private heavy hitters [8], private time-series database [27] and secure computation in the preprocessing model [13–15,20], showing significant speed-ups over previous approaches. In many of these settings, the class $\mathcal{F}$ supported by the FSS scheme corresponds to richer applications; for example, more sophisticated private database queries beyond private lookup.

Unfortunately, concretely efficient FSS constructions are only known for very limited function classes. For example, efficient function secret sharing schemes are known to exist for the class of point functions (i.e., functions that take a non-zero value only at a single input) and the class of comparison functions (i.e., functions that take the same non-zero value for all inputs less than a given point) [13,17,19]. While these are already sufficient for many powerful applications, they do not allow to support, for instance, complex database queries.

One way to obtain function secret sharing for richer classes of function is via *homomorphic secret sharing* (HSS) [18], the dual notion of function secret sharing, with the role of function and input reversed. HSS schemes for the class of polynomial-size branching programs (which in particular captures logarithmic-depth circuits) are known from a number of assumptions, such as the decisional Diffie-Hellman assumption [18], the DCR assumption [31,37,40], and the Learning With Errors assumption [22,29].

As observed in [18], there exists a generic transformation from a homomorphic secret sharing scheme to a function sharing scheme by relying on universal circuits. A universal circuit for a function class $\mathcal{F}$, is a circuit $C_{\mathcal{F}}$ such that

$\forall f \in \mathcal{F}, \forall x \in \{0,1\}^n$ it holds $C_{\mathcal{F}}(f,x) = f(x)$. Given such a universal circuit, one can transform the problem of constructing a function secret sharing scheme for $\mathcal{F}$ to the problem of constructing a homomorphic secret sharing scheme for the class of functions $\mathcal{C}_{\mathcal{F}} := \{C_{\mathcal{F}}(\cdot, x) \mid x \in \{0,1\}^n\}$.

For the class of branching programs, there exists a universal circuit that is itself a branching program [18]. Any homomorphic secret sharing scheme for the class of branching programs thus implies a function secret scheme for the same class. Unfortunately, the transformation introduces a high concrete overhead, especially when the structure of the branching program is wished to be hidden. More precisely, with the techniques given in [18], if w is an upper bound on the width of a binary branching program, then the resulting universal branching program has a blow-up of w^2 in depth, which leads to large key size and running time. For branching programs over larger fields, this overhead gets even worse. In fact, it is an open problem explicitly posed in a talk by Boyle [12, Page 85] to improve the efficiency of FSS over the universal branching program transformation.

We also consider deterministic finite automata (DFAs) in this work [39,43]. A DFA is an automaton with finitely many states that rejects or accepts a given string following a sequence of states, where the next state is determined by the next symbol of the string. As observed, e.g., in [34], if f is a function of input length n that is computed by a DFA with s states, it can be computed by a branching program of length n and size $s \cdot n + 1$, an FSS for branching programs thus directly yields an FSS for DFAs with bounded input-length. Note though that FSS for branching programs does not allow to compute general classes of DFAs, since these can support inputs of a-priori unbounded length, while yet having a succinct representation.

1.1 Our Contributions

In this work, we present constructions of function secret sharing schemes for the class of bit-fixing predicates, branching programs and more from an abstract pseudorandom generator with encoded-output homomorphism (EOH-PRG). We further show that if the EOH-PRG additionally satisfies a form of KDM-security, we can construct FSS for deterministic finite automata supporting inputs of a-priori *unbounded* length.

We give instantiations of the EOH-PRG from the standard learning with errors (LWE) assumption or a binary-secret variant of ring-LWE, as well as from a small-exponent variant of the decisional composite residuosity (DCR) assumption. We give an overview of the efficiency comparison of our concretely efficient FSS constructions for branching programs to previous FSS constructions via universal branching program transformations in Table 1. In terms of concrete efficiency, the run time is improved by at least a factor of w, where w is the width of the branching program.

In some sense, our work can be viewed as an extension of the line of work on exploring minicrypt primitives with algebraic structure and their applications, as started by Alamati et al. [2].

Table 1. Comparison of FSS for branching programs constructed from EOH-PRG and from HSS via universal branching programs. ℓ stands for the length of the branching program and w stands for the width of the branching program. Assume fixed out-degree $d = 2$. For the LWE assumption, n stands for the secret length, q the modulus of the LWE assumption, and p the output modulus of the PRG. The number of multiplications is counted over $\mathbb{Z}_q$. For the DCR assumption, N stands for RSA modulus. For the comparison with [37], we use their most efficient instantiation, for which they have to assume a DCR variant with circular security ([37, Section 4.2]). The number of exponentiations is counted over $\mathbb{Z}_{N^2}$.

		Assumption	Key Size	Run time(No. of Mul./ Exp.)
LWE	HSS [22]	Ring-LWE	$4\ell w^2 n \log q$	$8\ell w^2 n \log n$
	EOH-PRG(Ours)	Ring-LWE	$2\ell w(n+w)\log p$	$\ell(2+\lceil \frac{2w}{n} \rceil)n \log n$
DCR	HSS [37]	DCR	$7\ell w^2 \log N^2$	$14\ell w^2$
	EOH-PRG(Ours)	DCR	$2\ell w(w+1)\log N^2$	$\ell(3w+2)$

Our main results can be captured in a series of theorems. In the following, we will give a simplified definition of our EOH-PRG, which is yet too demanding for our instantiations, but allows to present the essence of our core theorems. For a full definition and more detailed explanation of our results, we refer to the technical overview section.

EOH-PRG. We start by introducing the concept of an EOH-PRG. Intuitively, an EOH PRG captures the functionality of a homomorphic pseudorandom generator in the following sense: Given an encryption $c = m + \mathsf{PRG}(s)$, a homomorphic PRG would allow to split the decryption key s into two shares $s_0 - s_1 = s$, s.t.,

$$(c_0 - \mathsf{PRG}(s_0)) - (c_1 - \mathsf{PRG}(s_1)) = m,$$

where $c_0 - c_1 = c$. In other words, a homomorphic PRG would allow the distributed decryption of m, where the size of the decryption keys s_0, s_1 are succinct (i.e., scale with the size of s, rather than $\mathsf{PRG}(s)$).

Unfortunately, perfectly homomorphic PRGs with both the domain and image being additive groups in the typical sense are not known to exist; one barrier is that any homomorphic PRG with an output space that supports efficient linear algebra can be broken by Gaussian elimination.

In this paper, we observe that if we relax the above to require the equation only relative to "encoded" messages m, it can be instantiated from standard assumptions.[1] We formalize this requirement in the following definition of pseudorandom generators with encoded output homomorphism. While this definition might look somewhat complex at first glance, we would like to stress that the intuition behind it is very simple: We leverage the observation that if we have

[1] Actually, for our instantiations we additionally have to restrict the seed s to be from a special subset $S \subset \mathbb{S}$, and our message from a special subset $H \subset \mathbb{H}$, but for simplicity we start by presenting our results with the slightly simpler definition.

some control over the message and PRG seed, one can recover the functionality of a homomorphic PRG while being able to give instantiations from standard assumptions.

Definition 1.1 (EOH-PRG, simplified). *Let* $\mathbb{S}, \mathbb{H}, \widetilde{\mathbb{H}}$ *be finite abelian groups. A function* $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ *is a PRG with encoded output homomorphism (EOH-PRG) relative to* $\mathbb{H}$ *if it is a pseudorandom generator and there exists a deterministic polynomial-time encoding function* $\mathsf{Encode} : \mathbb{H} \to \widetilde{\mathbb{H}}$ *and conversion (or "decoding") function* $\mathsf{Conv} : \widetilde{\mathbb{H}} \to \mathbb{H}$ *such that for all* $m \in \mathbb{H}$*, for* $s \in \mathbb{S}$ *it holds*

$$\mathsf{Conv}(c_0 - \mathsf{PRG}(s_0)) - \mathsf{Conv}(c_1 - \mathsf{PRG}(s_1)) = m,$$

where $c_0 - c_1 = \mathsf{PRG}(s) + \mathsf{Encode}(m)$ *and* $s_0, s_1 \in \mathbb{S}$ *with* $s_0 - s_1 = s$ *(except with negligible probability over the random choice of the shares).*

Note that given a truly homomorphic PRG, one could indeed instantiate the above definition of EOH-PRG by setting $\mathbb{H} := \widetilde{\mathbb{H}}$ and choosing Encode and Conv as identity functions.

We will show that the EOH-PRG can be instantiated with different paradigms: It can be instantiated by an *almost* homomorphic PRG, in which Conv corrects introduced errors and transforms shares in $\widetilde{\mathbb{H}}$ back to shares in $\mathbb{H}$ based on learning with errors (similar to the rounding and lifting in [22]), as well as with a homomorphic PRG mapping additive shares to multiplicative shares, in which Conv converts multiplicative shares back to additive shares based on a variant of the DCR assumption (similar to the conversion procedure in [18,37,40]), thereby presenting a way to unify these two approaches to distributed decryption.

For our constructions, we further need the PRG to support a "tag-space" $\mathbb{T}$. We will defer a formal definition to later, but we observe that for our constructions one can simply set $\mathbb{T} = \mathbb{Z}_\tau$, where τ is the order of $\mathbb{H}$ (which will also be satisfied by our instantiations).

Tensor Product Theorem. With this EOH-PRG, we can state our main results. We start by giving our tensor product theorem, which can be viewed as lifting the tensor product theorem of [19] for point predicates (i.e., the family of predicates taking 1 exactly at one point) to arbitrary predicates. Below we present it for the family of bit-predicates, we note though that it readily extends to any predicates with logarithmic-size input space.[2] For more detailed results we refer to the technical overview section and Sect. 5.

Theorem 1.1 (Tensor product FSS (simplified)). *Let* $\ell = \ell(\lambda)$ *be a polynomial. Let* $\mathcal{P}$ *be a family of predicates* $\{0,1\} \to \{0,1\}$*. Let* $\mathbb{S}, \widetilde{\mathbb{H}}, \mathbb{T}$ *be finite abelian*

[2] Note though that this assumes an EOH-PRG with an accordingly larger output space and thus results in larger key sizes.

groups. Then, if there exists an EOH-PRG $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ *relative to* $\mathbb{H} := (\mathbb{S}\times\mathbb{T})^2$ *with tag space* $\mathbb{T}$*, there exists an FSS for the function class*

$$\mathcal{P}^{\otimes} := \left\{ g_{P_1,\ldots,P_\ell}\colon \{0,1\}^\ell \to \{0,1\}, x \mapsto \bigwedge_{i=1}^{\ell} P_i(x_i) \;\middle|\; \forall i \in [\ell]\colon P_i \in \mathcal{P} \right\}$$

with polynomial key size.

By instantiating the above with the family of bit-fixing predicates, we obtain a FSS construction for bit-fixing predicate. We capture this result in the following corollary.

Corollary 1.1 (FSS for bit-fixing predicates). *Assume all parameters are as in Theorem 1.1 and* $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ *is a EOH-PRG relative to* $\mathbb{H} := (\mathbb{S} \times \mathbb{T})^2$ *with tag space* $\mathbb{T}$*. Then, there exists an FSS for* ℓ*-bit bit-fixing predicates with key size* $\log|\mathbb{H}| + (\ell-1)\log\left|\widetilde{\mathbb{H}}\right|$.

FSS for Branching Programs. Next, we state our main theorem for branching programs. We remark that the FSS for branching programs only hides the transition function whereas the topology of the branching program, i.e., the number of nodes of each level, is revealed. It is easy to extend each level to w nodes via adding dummy nodes and then construct an FSS for the extended branching program (note that the same has to be done in order to apply the generic transformation from HSS to FSS, if the topology is wished to be hidden). For more details on the FSS for branching programs, we refer to the technical overview section.

Theorem 1.2 (FSS for branching programs, simplified). *Let* P *be an oblivious, layered branching program with* ℓ *levels, width* w *and out-degree* d*. Let* $\mathbb{S}, \widetilde{\mathbb{H}}, \mathbb{T}$ *be finite abelian groups. Then, if there exists an EOH-PRG* $\mathsf{PRG} : \mathbb{S} \to \widetilde{\mathbb{H}}$ *relative to* $\mathbb{H} := (\mathbb{S} \times \mathbb{T}^w)^d$ *with tag space* $\mathbb{T}$*, there exists an FSS for* P *with key size* $\log|\mathbb{H}| + (\ell-1)\cdot w \cdot \log\left|\widetilde{\mathbb{H}}\right|$.

With FSS for branching programs, we present an FSS for the class of approximate matching functions in the full version [21]. We further give an FSS for multivariate polynomials over polynomial size rings in the full version.

FSS for DFAs. Finally, we give our construction of FSS for definite finite automata. Note that the construction of FSS for branching programs would directly imply an FSS for DFA, but requires the input size to be a-priori bounded as the FSS keys scale with the size of the input. Instead, we give a direct construction of a DFA, which can accept inputs of a-priori unbounded size (and for which the key sizes are independent of the size of the input). To that end, we introduce the notion of EOH-PRG with KDM-security. We stress that the kind of KDM-security we require for our FSS construction comes "for free" in our instantiations from LWE and DCR, without needing to assume a circular-security type assumption.

Definition 1.2 (KDM-secure EOH-PRG (simplified)). *Let Ψ be a family of embeddings $\psi\colon \mathbb{S} \to \mathbb{H}$. Let $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ be an EOH-PRG relative to $\mathbb{H}$. We say that PRG satisfies KDM-security relative to Ψ, if for each $\psi \in \Psi$, $\mathsf{PRG}^{\psi}(s) := \mathsf{PRG}(s) + \mathsf{Encode}(\psi(s))$ is a secure PRG.*

With this we obtain the following theorem.

Theorem 1.3 (FSS for DFAs (simplified)). *Let M be a DFA with state set Q and alphabet Σ. Let $\mu := |Q \cup \{A, R\}| = |Q| + 2$, where A and R stand for the merged accept state and rejection state, respectively. Let $\mathbb{S}, \widetilde{\mathbb{H}}, \mathbb{T}$ be finite abelian groups. Then, if there exists a EOH-PRG $\mathsf{PRG} : \mathbb{S} \to \widetilde{\mathbb{H}}$ relative to $\mathbb{H} := (\mathbb{S} \times \mathbb{T}^{\mu})^{|\Sigma|+1}$ with tag space $\mathbb{T}$ which satisfies KDM-security relative to a suitable function family Ψ, there exists an FSS for M with key size $|\mathbb{H}| + |Q| \cdot |\widetilde{\mathbb{H}}|$.*

It is worth to mention that the FSS for DFA is the first that allows key size independent of the length of the input(except for the generic constructions from FHE).

Towards Instantiating the EOH-PRG. In order to instantiate our constructions, we have to allow for a slightly more permissive notion of EOH-PRG, for which it is rather straightforward to adapt the above theorems. Namely, we additionally have to restrict the seed s to be from a special subset $S \subset \mathbb{S}$ (and require that the PRG restricted to S is still a PRG), and the message m from a special subset $H \subset \mathbb{H}$. With this relaxation, we show that it is possible to instantiate the EOH-PRG from LWE and binary-secret ring-LWE building on the techniques of [22], and from a short exponent variant of the DCR assumption inspired by the techniques of [37,40]. More precisely, we obtain the following results.

Theorem 1.4 (EOH-PRG from LWE (simplified)). *Let $n, p, q, r, \ell, w \in \mathbb{N}$ such that $r|p, p|q, 1 \ll r \ll p$,*[3] *and $n \log q < m \log p$, where $m := \ell(n + w)$. Further, let $q > 2pB$ and let χ be a B-bounded error distribution.*[4]

Then, assuming learning with errors $\mathsf{LWE}_{n,m,q,\chi}$ is hard, there exists an EOH-PRG $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ relative to $(S, H, \mathbb{H})$ with tag space $\mathbb{T}$, where $S = \{0,1\}^n, \mathbb{S} = \mathbb{Z}_p^n, \mathbb{T} = \mathbb{Z}_p, H = (S \times \{0,1\}^w)^{\ell} = \{0,1\}^m$ and $\widetilde{\mathbb{H}} = \mathbb{H} = (\mathbb{S} \times \mathbb{T}^w)^{\ell} = \mathbb{Z}_p^m$.

Recall that the DCR assumption states an N-th residue over $\mathbb{Z}_{N^2}^*$ is computationally indistinguishable from a random element over $\mathbb{Z}_{N^2}^*$. Based on the DCR assumption, Brakerski and Goldwasser [23] showed that $(g_1 \dots g_d, g_1^s \dots g_d^s)$ is pseudorandom, where $d \in \mathbb{N}$, each g_i is a N-th residue over $\mathbb{Z}_{N^2}^*$, and s is random element in $\mathbb{Z}_{\phi(N)}$. (Note that this can also be viewed as the DDH assumption over $\mathbb{Z}_{N^2}^*$.)

We have to rely on a variant of this assumption, where the secret is chosen from a (sufficiently large) bounded subspace $[-B/2, B/2] \subset \mathbb{Z}_{\phi(N)}$. Note that

[3] Here, by $\ll$ we denote a super-polynomial gap between parameters.

[4] Note that this requirement on the error distribution is to ensure that LWE implies LWR [7].

similar flavors of small-exponent assumptions have been used in [1,16,36]. With this, we obtain the following theorem.

Theorem 1.5 (EOH-PRG from DCR (simplified)). *Let B be an integer such that $B \cdot 2^{\lambda} \leq N$ and $B > 2^{\lambda}$. Further, let $\ell, w \in \mathbb{N}$ be arbitrary.*

Then, assuming a small exponent variant of DCR holds relative to B, there exists an EOH-PRG $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ *relative to $(S, H, \mathbb{H})$ with tag space $\mathbb{T}$, where $S = [-B/2, B/2], \mathbb{S} = \mathbb{Z}_{\phi(N^2)}, \mathbb{T} = \mathbb{Z}_{\phi(N^2)}, H = (S \times \{0,1\}^w)^{\ell}, \mathbb{H} = (\mathbb{S} \times \mathbb{T}^w)^{\ell} = (\mathbb{Z}_{\phi(N^2)})^{\ell(1+w)}$ and $\widetilde{\mathbb{H}} = (\mathbb{Z}^*_{N^2})^{\ell(1+w)}$.*

Note that in order for the DCR assumption to hold, the parties cannot know $\phi(N^2)$. In our construction, this will not be an issue. The computation mod $\phi(N^2)$ or $\phi(N)$ in the exponent is automatic because of the structure of the Paillier group, and to sample from $\mathbb{Z}_{\phi(N^2)}$, we can sample from $\mathbb{Z}_{N^2}$ instead, as the two distributions are statistically close. As we will explain in the technical overview, we are able to generate secret shares of elements x mod $\phi(N^2)$ whenever $|x|$ is sufficiently small (following the techniques of [37]), and can otherwise perform operations simply over $\mathbb{Z}$.

Comparisons. We give the concrete comparisons between our FSS for branching programs from EOH-PRGs and the previous FSS constructions via homomorphic secret sharing (HSS) in Table 1. Building on EOH-PRG yields more efficient constructions in terms of key size and runtime. Most notably, the new FSS schemes for branching programs provide significant improvements in run time over FSS from HSS for universal branching programs, by avoiding the overhead of the generic transformation. For example, consider the Multiply-Then-Truncate (MTT) operation [13], which is central for multiplying numbers in fixed-point arithmetic. With FSS for NC1, the MTT operation can be implemented in one round. The width for an oblivious BP for MTT is lower bounded by $w = N/logN$ [46] with N the input number length. For inputs of size N=64 bits as in [20], we thus obtain a lower bound $w = 10$ for the width of the BP. For the DCR-based instantiation we achieve an improvement of roughly a factor > 3.5 in the key size and factor > 40 in the run time and for the Ring-LWE based instantiation we obtain a factor around 20 improvement in the key size and a factor > 250 improvement in the run time. We want to highlight that the run time improvement both for the DCR and the LWE based instantiations scales with w (where w is the width of the BP), and thus is even more significant for wider branching programs. For details on the efficiency comparison we refer to the full version [21].

Applications. The central application of FSS schemes are forms of two-server private information retrieval [18]. Here, it is assumed that two (non-colluding) servers each hold a replication of a database DB with D items, and a client wants to launch a query to the database while keeping the query hidden from both servers individually. Given an FSS scheme supporting the query class, this can be achieved with succinct communication, by having the client split its query into succinct shares, which can then be evaluated by the server. By secrecy of the FSS, the servers do not learn anything about the query, as long as they are non-colluding.

In the full version [21], we show a number of applications including private image matching, private nearest neighbour search and private partial text matching, and how our construction can be used towards boosting the applications in terms of expressiveness and/or efficiency. In particular, our improved FSS constructions for bit-fixing and branching programs yield direct applications to applications such as 2-server private counting queries and private payload computations, as considered, e.g., in [22], with better efficiency.

Note that for the most part in our application we solely focus on achieving client privacy. If the underlying database contains privacy-critical data e.g., medical, biological or financial data, one further needs to consider server privacy. We show how this can be achieved in the example of private nearest neighbor search in the full version [21].

1.2 Discussion and Related Work

Beyond the Two-Party Case. Note that the FSS constructions from one-way functions [17,19,33] cannot be easily extended to more than two parties. Our FSS construction approach from EOH-PRGs, on the other hand, naturally extends beyond the two-party setting. However, it is not known how to instantiate the EOH-PRG from concrete assumptions for more than two parties. Our two-party instantiations from LWE and the DCR variant heavily rely on the distributed rounding [22] and distributed discrete logarithm [37], respectively, which were developed for two-party homomorphic secret sharing. To date, it is unclear how to generalize the distributed rounding or distributed discrete logarithm to more than two parties. In fact, [6] proved that there exists a barrier to directly generalize the share conversion from two-party to multi-party. Any such progress may lead to significant improvements for efficient multi-party FSS/HSS constructions.

On FSS from Weaker Assumptions. While constructing FSS for function classes such as branching programs solely based on the assumption of one-way functions would be a major breakthrough [17], it seems a more tractable open question if such FSS can be constructed for subclasses of $\mathtt{AC0}$ such as bit-fixing predicates or t-CNF. In the technical overview, we give some intuition why it seems unlikely that the techniques of the line of work on FSS from one-way functions [17,19,33] allow for this without relying on additional structure (such as EOH-PRGs), due to an inherent exponential blow-up. An alternative route could be taken following [28], who give constructions of privately constrained PRFs for t-CNFs from one-way functions. Here, however, the problem is that de-randomizing the constrained points to fixed values would again introduce an exponential blow-up. We leave it as an interesting open questions to either give such candidates, or give barriers towards their construction.

Relation to Secure Branching Program Evaluation Protocols. There is a line of work on secure branching program evaluation (BPE) [4,5,11,24,35,44,47] relying on garbled circuits or homormorphic encryption. The setting considered in their work is somewhat orthogonal to ours: They consider a branching program (held by a sender) to be evaluated on a single input (held by a receiver), such that the result is learned by the receiver, and such that both the branching program provided by the sender and receiver input remain hidden. We, on the other hand, consider a branching program (held by a client) to be evaluated on a database (held by two servers), such that a linear combination of the outputs is learned by the client, and such that the branching program (i.e., database query) provided by the client remains hidden, as long as the two servers are not colluding. With our approach, the communication cost scales with $\log N$ for a database of size N, since the same branching program can be evaluated on *all* inputs. Except for the FHE-based approach [11], the communication cost of all other protocols in the BPE line of work instead scales with N to achieve the same functionality. This is even true for the protocols [47] relying on additively homomorphic encryption, since they still require communication between the receiver and sender *per input* to be evaluated. It is worth to point out that sublinear communication complexity in the line of work on BPE (as achieved in [44]) refers to sublinear in the *size of the branching program*, whereas we consider settings where the size of the database N is the dominating cost.

1.3 Organization

Only the main results and techniques are presented in the body part. Section 2 presents an overview of the central techniques, followed by preliminaries in Sect. 3. The EOH-PRG is formally defined in Sect. 4. We show the constructions for tensor product, branching programs and DFAs in Sect. 5, 6, 7, respectively. Finally, in Sect. 8 we present instantiations of the EOH-PRG.

2 Technical Overview

In the following we give an overview of the central techniques. We start by explaining the *tensor product FSS for point functions* of Boyle, Gilboa and Ishai [19] (in the following refered to as BGI16), and show how to extend their construction to a more general tensor product using an encoded-output PRG. Then, we show how this yields FSS for the classes of bit-fixing predicates. Next, we explain how the construction can be extended towards FSS for branching programs and for DFAs.

Background [17,19]. Before giving the construction, we recall some required preliminaries. Firstly, recall that a *point function* is simply a function that takes a non-zero value only at one dedicated point. More precisely, the point function

f_α^β with input space $\{0,1\}^n$ and output space $\mathcal{R}$ (for some group $\mathcal{R}$) is defined as

$$f_\alpha^\beta(x) := \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{else} \end{cases}.$$

A *function secret sharing scheme* for a family of function $\mathcal{F}$ consists of tuple of PPT algorithms $(\mathsf{Gen}, \mathsf{Eval})$, such that Gen takes as input the description $\hat{f}$ of f and returns a tuple of keys (k_0, k_1) and Eval takes as input a party index b, a key k_b and an input value x and outputs an output value y_b, such that the following holds:

Correctness: For all $x \in \{0,1\}^n$. it holds $\mathsf{Eval}(0, k_0, x) - \mathsf{Eval}(1, k_1, x) = f(x)$.
Secrecy: For $b \in \{0,1\}$, k_b computationally hides $\hat{f}$ within $\mathcal{F}$.

Note that an FSS for the class of point functions, is also refered to as *distributed point function (DPF).*

Tensor Product FSS for Point Functions [BGI16 [19]]. Given a function secret sharing scheme for the class $\mathcal{F}^\circ$ of *point functions*, and a function secret sharing scheme for a function class $\mathcal{F}$ of *arbitrary functions*, BGI16 gives a construction for the *tensor product* $\mathcal{F}^\circ \otimes \mathcal{F}$, i.e., the class of functions

$$F_{\alpha,f}(x_1, x_2) := \begin{cases} f(x_2) & \text{if } x_1 = \alpha \\ 0 & \text{else} \end{cases}$$

for $\alpha \in \{0,1\}^n, f \in \mathcal{F}$, where the key size scales polynomially in the key sizes of the underlying FSS schemes.[5]

In the following, we describe the construction of BGI16 in a number of steps, adding layers of secrecy one-by-one. For the construction we assume that the FSS scheme $(\mathsf{Gen}, \mathsf{Eval})$ for $\mathcal{F}$ satisfies a *symmetry* property, i.e., $\mathsf{Eval}(0, k, x) = \mathsf{Eval}(1, k, x)$ for all inputs x and keys k. Further, we assume that keys $(k_0, k_1) \leftarrow \mathsf{Gen}(f)$ are individually pseudorandom over the same key space $\mathcal{K}$.

First Attempt: A Construction with Very Limited Secrecy: To get a construction where α is hidden from P_1, one can proceed as follows: To share $F_{\alpha,f}$, one can generate keys $(k_0, k_1) \leftarrow \mathsf{Gen}(\hat{f})$ and set $K_0 := (\alpha, k_0, k_1)$ and $K_1 := k_1$. To evaluate on point (x_1, x_2), party P_0 outputs $y_0 := \mathsf{Eval}(0, k_0, x_2)$ if $x_1 = \alpha$ and $y_0 := \mathsf{Eval}(0, k_1, x_2)$ otherwise. Party P_1 simply outputs $y_1 := \mathsf{Eval}(1, k_1, x_2)$.

Correctness and very limited secrecy: Here, for $x_1 = \alpha$ we have $y_0 - y_1 = \mathsf{Eval}(0, k_0, x_2) - \mathsf{Eval}(1, k_1, x_2) = f(x_2)$ by the correctness of $(\mathsf{Gen}, \mathsf{Eval})$. For $x_1 \neq \alpha$, on the other hand, it holds $y_0 - y_1 = \mathsf{Eval}(0, k_1, x_2) - \mathsf{Eval}(1, k_1, x_2) = 0$

[5] The resulting scheme actually satisfies a stronger notion of key compactness, namely the non-public part of the key does not grow, allowing to apply the tensor product operation recursively a polynomial number of times.

by symmetry, as required.[6] This construction does obviously hide α from P_1, but otherwise does not provide any secrecy guarantees.

Second Attempt: A Construction with Secret α. Towards hiding α also from P_0, the trick is to additionally use the FSS scheme $(\mathsf{Gen}^\circ, \mathsf{Eval}^\circ)$ for $\mathcal{F}^\circ$, and flipping the order of k_0 and k_1 with probability 1/2, thereby hide from the parties when they use the same keys. More precisely, assume to be given an FSS for point functions with output space $\{0,1\}$ (i.e., the point function maps to 1 at the unique non-zero point α, and otherwise to 0). Now, the idea is to generate keys $(k_0^\circ, k_1^\circ) \leftarrow \mathsf{Gen}^\circ(\hat{f}_\alpha^1)$, and compute "tag" values $\tau_b \leftarrow \mathsf{Eval}^\circ(b, k_b^\circ, \alpha)$ (i.e., $\tau_0 \oplus \tau_1 = 1$ by construction) relative to these keys. The tag values are used to hide if the parties use the same key k_b, by defining $\mathsf{cw}_{\tau_b} := k_b$ (where $(k_0, k_1) \leftarrow \mathsf{Gen}(\hat{f})$ as before) and setting $K_b := (k_b^\circ, \mathsf{cw}_0, \mathsf{cw}_1)$. To evaluate at a point (x_1, x_2), party P_b first computes $t_b \leftarrow \mathsf{Eval}^\circ(b, k_b^\circ, x_1)$ and then outputs $y_b \leftarrow \mathsf{Eval}(b, \mathsf{cw}_{t_b}, x_2)$.

Correctness and secrecy of α: Now, for $x_1 = \alpha$, it holds $\mathsf{cw}_{t_b} = \mathsf{cw}_{\tau_b} = k_b$. As before, the parties thus obtain $y_0 - y_1 = \mathsf{Eval}(0, k_0, x_2) - \mathsf{Eval}(1, k_1, x_2) = f(x_2)$ as required. If $x_1 \neq \alpha$, on the other hand, it holds $t_0 = t_1$ and thus $k_{t_0} = k_{t_1}$, implying $y_0 - y_1 = \mathsf{Eval}(0, k_{t_0}, x_2) - \mathsf{Eval}(1, k_{t_0}, x_2) = 0$, again by symmetry. The construction hides α from both parties by the secrecy of $(\mathsf{Gen}^\circ, \mathsf{Eval}^\circ)$ and the pseudorandomness of keys for $(\mathsf{Gen}, \mathsf{Eval})$ (which prevents the parties from learning when they use the real key at position α and when they use a "dummy key"), but still fully leaks f.

The Construction of BGI16. The idea of BGI16 to overcome this, is to additionally use a pseudorandom generator to *blind* the keys k_0, k_1, such that party P_0 is only able to recover k_0 and party P_1 is only able to recover k_1 at the dedicated point $x_1 = \alpha$ (without being able to distinguish this from the case where both parties recover the same "dummy" key, to ensure that α remains hidden). To this end, assume that FSS° is now an FSS for point functions with output space $\{0,1\}^{\lambda+1}$. The idea of BGI16 is as follows: To generate a key for $F_{\alpha,f}$, the key generation algorithm starts by choosing $s \leftarrow_R \{0,1\}^\lambda$ at random and generating $(k_0^\circ, k_1^\circ) \leftarrow \mathsf{Gen}^\circ(\hat{f}_\alpha^{s,1})$. Further, the key generation algorithm generates the corresponding "seed values" $\sigma_b \in \{0,1\}^\lambda$ and, again, "tag values" $\tau_b \in \{0,1\}$ as $(\sigma_b, \tau_b) \leftarrow \mathsf{Eval}^\circ(b, k_b^\circ, \alpha)$ (i.e., $\sigma_0 \oplus \sigma_1 = s$ and $\tau_0 \oplus \tau_1 = 1$ by construction). Further, given a pseudorandom generator $\mathsf{PRG} : \{0,1\}^\lambda \to \mathcal{K}$, the full "correction words" are generated as $CW_{\tau_b} := k_b + \mathsf{PRG}(\sigma_b)$ (where $(k_0, k_1) \leftarrow \mathsf{Gen}(\hat{f})$ as before) and the keys defined as $K_b := (k_0^\circ, CW_0, CW_1)$. To evaluate on point (x_1, x_2), the parties now compute $(s_b, t_b) \leftarrow \mathsf{Eval}^\circ(b, k_b^\circ, x_1)$, "correct" their keys to $\kappa_b := CW_{t_b} - \mathsf{PRG}(s_b)$ and evaluate to $y_b \leftarrow \mathsf{Eval}(b, \kappa_b, x_2)$.

Correctness and secrecy of BGI16: If $x_1 = \alpha$, it holds $\kappa_b = CW_{t_b} - \mathsf{PRG}(s_b) = CW_{\tau_b} - \mathsf{PRG}(\sigma_b) = k_b$ and thus $y_0 - y_1 = \mathsf{Eval}(0, k_0, x_2) - \mathsf{Eval}(1, k_1, x_2) = f(x_2)$ as required. If $x_1 \neq \alpha$, on the other hand, then $t_0 = t_1$ and thus $\kappa_0 = \kappa_1$ (i.e.,

[6] This construction would not actually require symmetry of the underlying FSS since P_0 knows when $x_1 \neq \alpha$ and could evaluate $\mathsf{Eval}(1, k_1, x_2)$in this case, but this will no longer be possible in the subsequent constructions.

both parties recover the same "dummy" key), and $y_0 - y_1 = \mathsf{Eval}(0, \kappa_0, x_2) - \mathsf{Eval}(1, \kappa_0, x_2) = 0$ by symmetry. Full secrecy holds by the above considerations and because $(\mathsf{Gen}, \mathsf{Eval})$ satisfies secrecy and pseudorandomness of keys, which prevents the parties from learning where the true FSS keys are embedded.

Limitation of BGI16 to Tensoring with Point Functions. The issue with extending the above approach even slightly beyond point functions (e.g., to function which take a non-zero value at two points) is that it would incur an exponential blow-up in the key size (and run time of the key generation), since the parties have to recover *different* key pairs (K_0, K_1) and (K_0', K_1') for different non-zero points α, α' (as reusing a key would allow the parties to locally derive information about the position of non-zero points). Note that this includes the "correction word" part CW_0, CW_1 of the key, since keys with different first component require different correction words in construction of BGI16. The key generation time and key length thus (at least) *double* at each tensoring. Recursive tensoring is therefore limited to at most a logarithmic number of times, which is not sufficient for most applications.

This issue could be overcome, if the keys could be "randomized" in order to hide that the *same* key is reused. For additive secret sharing this is trivially the case: Namely, assume an output value is shared as $y = k_0 - k_1 \in \mathcal{K}$ (for some additive group $\mathcal{K}$). Then, for any $\Delta \in \mathcal{K}$, the secret can be re-shared as $(k_0 + \Delta, k_1 + \Delta)$, which looks like perfectly fresh keys from the view of the adversary. Unfortunately, the construction of BGI16 does not satisfy this property of "shift-invariance", even if the underlying FSS schemes $\mathcal{F}^\circ$ and $\mathcal{F}$ were to satisfy these properties: Namely, even if $\sigma_0 - \sigma_1 = \sigma_0' - \sigma_1'$ (where $\sigma_b \leftarrow \mathsf{Eval}^\circ(b, k_b, \alpha)$ and $\sigma_b' \leftarrow \mathsf{Eval}^\circ(b, k_b, \alpha')$), the PRG outputs $\mathsf{PRG}(\sigma_b)$ and $\mathsf{PRG}(\sigma_b')$ are in general uncorrelated.

This could be resolved by using an ideal *homomorphic* PRG, ensuring that the correlation is preserved to $\mathsf{PRG}(\sigma_0) - \mathsf{PRG}(\sigma_1) = \mathsf{PRG}(\sigma_0') - \mathsf{PRG}(\sigma_1')$. Unfortunately, perfectly homomorphic PRGs with both the domain and image being additive groups in the typical sense are not known to exist. For simplicity, we still start by outlining our tensor product construction assuming access to a perfectly homomorphic PRG $\mathsf{PRG}\colon \{0,1\}^\lambda \to \mathcal{K}$, before giving our full construction.

Overcoming the Limitations via an Ideal Homomorphic PRG. We start by simplifying the construction of BGI16 assuming access to a perfectly homomorphic PRG $\mathsf{PRG}\colon \{0,1\}^\lambda \to \mathcal{K}$, and assuming a *shift-invariant* FSS $\mathsf{FSS} = (\mathsf{Gen}, \mathsf{Eval})$ for $\mathcal{F}$ with key space $\mathcal{K}$,[7] i.e., for $(k_0, k_1) \leftarrow \mathsf{Gen}(\hat{f})$, we assume that any shifted tuple $(k_0 + \Delta, k_1 + \Delta)$ for $\Delta \in \mathcal{K}$ constitutes a valid key pair for f. Then, the construction of BGI16 can be simplified to a construction requiring only one correction word CW:

[7] Note, that for correctness of the simplified construction outlined below, we would actually require $(\mathcal{K}, +) := (\{0,1\}^k, \oplus)$, for some $k \in \mathcal{K}$. To be aligned with the general construction, we will still use th notation $(\mathcal{K}, +)$ in the following.

Again, to generate a key for $F_{\alpha,f}$, the key generation algorithm samples $s \leftarrow_R \{0,1\}^\lambda$ and generates keys $(k_0^\circ, k_1^\circ) \leftarrow \mathsf{Gen}^\circ(f_\alpha^{s,1})$. Instead of pre-computing the tag and seed values at position α, the key generation algorithm simply generates $(k_0, k_1) \leftarrow \mathsf{Gen}(\hat{f})$, sets $CW := k_0 - k_1 + \mathsf{PRG}(s)$ and outputs (K_0, K_1), where $K_b := (k_b^\circ, CW)$. To evaluate, the parties now compute $(s_b, t_b) \leftarrow \mathsf{Eval}^\circ(b, k_b^\circ, x_1)$ and then obtain the "corrected" keys as $\kappa_b := t_b \cdot CW - \mathsf{PRG}(s_b)$. (Note that $t_b \in \{0,1\}$, and thus the multiplication simply corresponds to adding 0 or $\mathcal{CW}$.)

Correctness holds, since at position α it holds $s_0 - s_1 = s$ and $t_0 - t_1 = 1$, and thus[8]

$$\kappa_0 - \kappa_1 = (t_0 \cdot CW - \mathsf{PRG}(s_0)) - (t_1 \cdot CW - \mathsf{PRG}(s_1)) = CW - \mathsf{PRG}(s) = k_0 - k_1.$$

As FSS is shift-invariant, the above implies that (κ_0, κ_1) and (k_0, k_1) are functionally equivalent, and thus $y_0 - y_1 = \mathsf{Eval}(0, \kappa_0, x_2) - \mathsf{Eval}(1, \kappa_1, x_2) = f(x_2)$ as required. If $x_1 \neq \alpha$, on the other hand, we obtain $s_0 = s_1$ and $t_0 = t_1$ and thus $\kappa_0 = \kappa_1$ as before, and correctness follows from the symmetry of FSS. Secrecy holds by the secrecy of the underlying FSS schemes, together with the pseudorandomness of PRG.

Note that this simplified scheme readily extends beyond point functions.

EOH-PRG. In order to instantiate the above construction, we introduce the notion of *PRG with encoded-output homomorphism* (EOH-PRG) and show that the above construction can be extended to support instantiation from this weaker notion. Roughly, an EOH-PRG has "encoding" and "conversion" (or "decoding") functions Encode and Conv such that it satisfies the following: given additive secret shares (s_0, s_1) of a seed s, and additive secret shares (y_0, y_1) of a blinded encoding $\mathsf{PRG}(s) + \mathsf{Encode}(m)$, we require $\mathsf{Conv}(y_0 - \mathsf{PRG}(s_0)) - \mathsf{Conv}(y_1 - \mathsf{PRG}(s_1)) = m$ (except with negligible probability over the random choice of the secret shares). Intuitively, this is sufficient to instantiate (a variant of) the tensor product FSS above, by encoding the key difference $k_0 - k_1$ to $\mathsf{Encode}(k_0 - k_1)$ before adding $\mathsf{PRG}(s)$.

More formally, a EOH-PRG $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ as required for our tensor product construction is parametrized by $S, H, \mathbb{H}$, together with (efficiently computable) maps $\mathsf{Encode}\colon H \to \widetilde{\mathbb{H}}$ and $\mathsf{Conv}\colon \widetilde{\mathbb{H}} \to \mathbb{H}$ such that:

- $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ is a PRG.
- $S \subset \mathbb{S}$ is such that PRG restricted to S is still a PRG and $0 \in S$. Note that S will serve as the seed space of our tensor product (recall that before $S = \{0,1\}^\lambda$). 0 has to be included in S to account for the case where both parties recover the same "dummy" seed value $s_0 = s_1$, i.e., $s_0 - s_1 = 0$.
- $\mathbb{H}$ is a finite abelian group containing the set $H \subset \mathbb{H}$. Note that $\mathbb{H}$ will correspond to the key space $\mathcal{K}$ of the FSS FSS before encoding (i.e., recovery of the keys is relative to addition in $\mathcal{K}$). $H \subset \mathbb{H}$ will correspond to the set

[8] Note that to obtain $t_0 \cdot CW - t_1 \cdot CW = CW$ we use $(\mathcal{K}, +) = (\{0,1\}^k, \oplus)$. We will later show how to generalize this.

of actual key differences $k_0 - k_1$ for $(k_0, k_1) \leftarrow \mathsf{Gen}(\hat{f})$ for $f \in \mathcal{F}$. Having separate $H \subset \mathbb{H}$ stems from the intantiations of EOH-PRG – given a truly homomorphic PRG, one could simply choose $H = \mathbb{H} = \widetilde{\mathbb{H}}$.
- $\widetilde{\mathbb{H}}$ is a finite abelian group containing the image of the PRG.

Finally, $\mathsf{Enc}\colon H \to \widetilde{\mathbb{H}}$ and $\mathsf{Conv}\colon \widetilde{\mathbb{H}} \to \mathbb{H}$ are such that for all $s \in S$, for all $m \in H$, for random secret shares $s_0, s_1 \leftarrow_R \mathbb{S}$ with $s_0 - s_1 = s$, and for random secret shares $y_0, y_1 \leftarrow_R \widetilde{\mathbb{H}}$ with $y_0 - y_1 = \mathsf{PRG}(s) + \mathsf{Encode}(m)$ it holds

$$\mathsf{Conv}(y_0 - \mathsf{PRG}(s_0)) - \mathsf{Conv}(y_1 - \mathsf{PRG}(s_1)) = m$$

in $\mathbb{H}$ except with negligible probability.

Further, we require a tag space which operates on $\widetilde{\mathbb{H}}$. More formally, we require the following:

- $\mathbb{T}$ is a finite abelian group containing $\{0,1\} \subset \mathbb{T}$. Note that $T = \{0,1\}$ will serve as the tag space of our tensor product FSS, which is embedded in the group $\mathbb{T}$, i.e., recovery of the tag will now be additive over $\mathbb{T}$, rather then additive over $(\{0,1\}, \oplus)$.
- $\cdot\colon \mathbb{T} \times \widetilde{\mathbb{H}} \to \widetilde{\mathbb{H}}$ is an efficiently computable non-trivial (left) homomorphic group operation of $\mathbb{T}$ on $\widetilde{\mathbb{H}}$, i.e., $\forall t_0, t_1 \in \mathbb{T}$ and $\forall h \in \widetilde{\mathbb{H}}$, $t_0 \cdot h + t_1 \cdot h = (t_0 + t_1) \cdot h$. Note that we need to define an operation of $\mathbb{T}$ on $\widetilde{\mathbb{H}}$ in order to "correct" the keys based on the tag values. This allows to support more general key spaces than $\mathcal{K} = \{0,1\}^k$.

As mentioned above, In the following, we will assume this as part of the definition of an EOH-PRG and simply add $\mathbb{T}$ to the parametrization.

A First Step: Tensor Product FSS for Arbitrary Predicates from EOH-PRG. Our tensor product construction from a EOH-PRG is essentially the same as the one from a perfectly homomorphic PRG (assuming the underlying FSS satisfy some additional properties), except that the correction word is computed as $CW := \mathsf{PRG}(s) + \mathsf{Enc}(k_0 - k_1)$, and the keys are recovered as $\kappa_b := \mathsf{Conv}(t_b \cdot CW - \mathsf{PRG}(s_b))$.[9]

Note that for construction to satisfy correctness, it is now required that the FSS scheme FSS for $\mathcal{F}$ satisfies $k_0 - k_1 \in H$ for all $f \in \mathcal{F}$ and $(k_0, k_1) \in \mathsf{Gen}(\hat{f})$ (since the encoding function Encode takes inputs in H), but satisfies shift invariance relative to the additive group $\mathbb{H}$ (since the decoding function Conv returns elements in $\mathbb{H}$).

[9] A subtlety is that in the definition of EOH-PRG we only require correctness relative to random shifts, but here we rely on correctness relative to shifts of the form $t_b \cdot CW - \mathsf{PRG}(s_b)$. This can be solved by having the parties re-randomize their shares with random offset $\mathsf{PRF}(s, i)$ (where each time a fresh index i is used and both parties hold the key s). This is necessary anyway for applying tensoring recursively, and allows for a simpler definition of EOH-PRG. Note though that this requires us to settle with a form of "non-adaptive" correctness as used e.g. in [22], where the inputs are assumed to be chosen independently of the keys.

An example for such an FSS can be obtained by additively secret-sharing the truth table with values in H over $\mathbb{H}$ if the function class is sufficiently small, this can be done efficiently.

With this we obtain the following theorem.

Theorem 2.1 (Theorem 5.1). *Assume* $\mathsf{PRG} : \mathbb{S} \to \tilde{\mathbb{H}}$ *is a EOH-PRG parametrized by* $(S, \mathbb{T}, H, \mathbb{H})$*. Further, let* $\mathsf{FSS}^{\mathcal{P}} = (\mathsf{Gen}^{\mathcal{P}}, \mathsf{Eval}^{\mathcal{P}})$ *be an FSS for a function family* $f_P^\beta \colon \{0,1\}^{n_1} \to \mathbb{S} \times \mathbb{T}$*, where*

$$f_P^\beta(x_1) = \begin{cases} \beta & \text{if } P(x_1) = 1 \\ 0 & \text{else} \end{cases},$$

for $\beta \in S \times \{1\} \subseteq S \times T \subseteq \mathbb{S} \times \mathbb{T}$ *(i.e., recovery is additive in* $\mathbb{S} \times \mathbb{T}$*) and* $P \in \mathcal{P}$*, and let* $\mathsf{FSS} = (\mathsf{Gen}, \mathsf{Eval})$ *be a symmetric and shift-invariant FSS for some class of functions* $\mathcal{F}$ *of the form* $f \colon \{0,1\}^{n_2} \to \mathcal{R}$ *with key space* $\mathcal{K} = \mathbb{H}$*, such that for any pair of keys* $(k_0, k_1) \leftarrow \mathsf{Gen}(\hat{f})$ *it holds* $k_0 - k_1 \in H$*.*

Then, there exists an FSS $\mathsf{FSS}^{\otimes} = (\mathsf{Gen}^{\otimes}, \mathsf{Eval}^{\otimes})$ *for the class* $\mathcal{F}^{\mathcal{P}} \otimes \mathcal{F}$ *of functions*

$$F_{P,f} \colon \{0,1\}^{n_1+n_2} \to \mathbb{G},\ (x_1, x_2) \mapsto \begin{cases} f(x_2) & \text{if } P(x_1) = 1 \\ 0 & \text{else} \end{cases},$$

for $P \in \mathcal{P}, f \in \mathcal{F}$*, where the resulting keys consist of a "secret" part corresponding to the key space in* $\mathsf{FSS}^{\mathcal{P}}$ *and a "correction word" space* $\tilde{\mathbb{H}}$*.*

Theorem 2.1 yields the following corollary.

Corollary 2.1 *Assume* $\mathsf{PRG} : \mathbb{S} \to \tilde{\mathbb{H}}$ *is a EOH-PRG relative to* $(S, \mathbb{T}, H, \mathbb{H})$ *with* $H = (S \times T)^2, \mathbb{H} = (\mathbb{S} \times \mathbb{T})^2$*. Assume* $\mathcal{P}$ *is a family of predicates* $\{0,1\} \to \{0,1\}$*. Then, there exists an FSS for the function class*

$$F_{P_1 \wedge \cdots \wedge P_\ell} \colon \{0,1\}^\ell \to \{0,1\},\ \mathbf{x} \mapsto \bigwedge_{i=1}^{\ell} P_i(\mathbf{x}[i]),$$

where $P_1, \ldots, P_\ell \in \mathcal{P}$*.*

FSS for Branching Programs. In general, a branching program can be described as a finite directed acyclic graph with one source node and two sink nodes, *accept* and *reject*. In this section, we focus on giving an FSS construction for oblivious ℓ-layered branching programs with out-degree 2, i.e., each width-w layer uses a fixed position of the input, each non-sink node has two outgoing edges labeled by 0 and 1, and edges only go from one level to the next level, as specified by a transition function $f \colon [\ell] \times [w] \times \{0,1\} \to [w]$.

Roughly, the idea to obtain an FSS for the class of such branching programs is to extend the tag value $t \in \{0,1\}$ to a *tag vector* $\mathbf{t} \in \{0,1\}^w$, allowing to pick the "right" correction word in going from the i-th to the $(i+1)$-th level.

More precisely, for the i-th level, the key generation algorithm essentially chooses a seed $\mathbf{s}_i \in S^w$, i.e., the j-th node on the i-th level is "labelled" by a seed value $s_{i,j} \in S$ together with a fixed tag vector $\mathbf{e}_j \in \{0,1\}^w$, where $\mathbf{e}_j$ corresponds to the j-th unit vector. Recall that each node has two outgoing edges, 0 and 1. In other words, for each node $(s_{i,j}, \mathbf{e}_j)$ on level i, there exist two possible nodes $j_0 := f(i,j,0)$ and $j_1 := f(i,j,1)$ that can be reached in level $i+1$ with corresponding labels $(s_{i+1,j_0}, \mathbf{e}_{j_0})$ and $(s_{i+1,j_1}, \mathbf{e}_{j_1})$. To go from the i-th to the $(i+1)$-th level, the idea is now to encode the transitions into correction words. More precisely, we define

$$CW_i[j] := \mathsf{PRG}\left(s_{i,j}\right) + \mathsf{Encode}\left((s_{i+1,j_0}, \mathbf{e}_{j_0}), (s_{i+1,j_1}, \mathbf{e}_{j_1})\right).$$

Thus, the correction word for each node can be computed in this way. The FSS key consists of a sharing of the label of start node and correction words.

During evaluation, the tag vector allows to pick the right correction word to proceed to the next level. Namely, given secret shares $(s_b, \mathbf{t}_b)$ such that

$$(s_0, \mathbf{t}_0) + (s_1, \mathbf{t}_1) = (s_{i,j}, \mathbf{e}_j)$$

the parties can compute

$$y_b = \mathsf{Conv}\left(\sum_{k=1}^{w} \mathbf{t}_b[k] \cdot CW_i[k] - \mathsf{PRG}(s_b)\right)$$

to obtain

$$\begin{aligned} y_0 - y_1 = {} & \mathsf{Conv}\left(\sum_{k=1}^{w} \mathbf{t}_0[k] \cdot CW_i[k] - \mathsf{PRG}(s_0)\right) \\ & - \mathsf{Conv}\left(\sum_{k \in [w_i]} \mathbf{t}_1[k] \cdot CW_i[k] - \mathsf{PRG}(s_1)\right) \\ = {} & \mathsf{Conv}(\mathbf{t}_0[j] \cdot CW_i[j] - \mathsf{PRG}(s_0)) + \mathsf{Conv}(\mathbf{t}_1[j] \cdot CW_i[j] - \mathsf{PRG}(s_1)) \\ = {} & \left((s_{i+1,j_0}, \mathbf{e}_{j_0}), (s_{i+1,j_1}, \mathbf{e}_{j_1})\right), \end{aligned}$$

by the property of the EOH-PRG. The parties can now continue the evaluation with the left or right part of the output, depending on the i-th input bit.

Note that in the described inductive construction of FSS for branching programs, we assumed that each level has exactly w nodes. This can be achieved by virtually adding dummy nodes, for which the correction words for dummy nodes can be sampled uniformly at random, since these are never reached during evaluation.

Comparison with FSS via Universal Branching Programs. Previous constructions of FSS for branching programs rely on homomorphic secret sharing (HSS) [18, Theorem 4.15] or FSS for all functions from fully homomorphic encryption(FHE) [29, Section 6.3]. Given a branching program P, the construction of

[18] encodes P to $\hat{P}$ and generates a universal branching program (UBP) for P such that $UBP(\hat{P}, x) = P(x)$ for arbitrary x. Next, $\hat{P}$ is secret-shared via the underlying HSS to hide the transition function of P. Note that the transformation via UBPs incurs a considerate efficiency blow-up which is at least quadratic in the number of levels. Refer to the full version [21] for details.

In contrast, the evaluation of our FSS construction directly emulates the evaluation of the branching program. For each level of the program P, the PRG needs to be evaluated once. Moreover, our FSS for branching programs naturally supports multi-edges. For the FSS via universal branching programs, on the other hand, the multi-edges need to be splitted into plain edges, incurring an additional blow-up. Finally, for branching programs with polynomial out-degree, say d, our FSS construction only increases the correction word for each node from two elements to d elements.

FSS for DFAs. Given a DFA $M := (Q, \Sigma, \delta, q_0, F)$, we first transform the set of accepting states F to a single accept state A via appending a special symbol ϵ to the end of each input. Similarly, we can transfer the remaining states to a rejection state R. The resulting DFA has alphabet $\Sigma \cup \{\epsilon\}$ and states set $Q \cup \{A, R\}$.

Similarly to the FSS for branching programs, the FSS for DFAs focuses on hiding the transition function. For each state $s \in Q$, the label for s consists of a uniformly random seed and a tag vector assigned according to a designated order of the states. The correction word for s hides the labels of one-step reachable states from s. The key for the FSS is a random sharing of the label for q_0 and the correction word for every state in Q. Since a state may be transferred to itself via some symbols, a KDM secure variant of EOH-PRG is necessary. With EOH-PRG, the key size of this FSS is independent of the length of string to be evaluated by the DFA.

Instantiating the EOH-PRG. In the following we explain our instantiations of EOH-PRG from LWE and a small-exponent DCR variant.

EOH-PRG from LWE. Recall that the LWE assumption naturally provides an almost-homomorphic PRG (AH-PRG). Let $p, q \in \mathbb{N}$ with $p|q$ and let $\lceil\cdot\rfloor_{q\to p}$ be defined as $\lceil\cdot\rfloor_{q\to p} : \mathbb{Z}_q \to \mathbb{Z}_p, x \mapsto \lceil(p/q)\cdot x\rfloor$. Suppose $A \leftarrow_R \mathbb{Z}_q^{n\times m}$. Then, $\mathsf{PRG}_A : \mathbb{Z}_q^n \to \mathbb{Z}_p^m, \mathbf{s} \mapsto \lceil \mathbf{s}A\rfloor_{q\to p}$ is an AH-PRG [10]. The security of PRG_A follows from the pseudorandomness of LWR distributions and the almost homomorphic property naturally follows from the rounding operation. It is easy to verify that

$$\mathsf{PRG}_A(\mathbf{s}_0 + \mathbf{s}_1) = \mathsf{PRG}_A(\mathbf{s}_0) + \mathsf{PRG}_A(\mathbf{s}_1) + \mathbf{e}$$

with $\|\mathbf{e}\|_\infty \le 1$.

Note though that an AH-PRG is not sufficient to instantiate our construction, since the small error vector would lead to correctness errors with too high probability. To overcome this problem and obtain an EOH-PRG, we rely on the distributed rounding and lifting technique as introduced in [22]. Concretely, let $r \in \mathbb{N}$ be an integer such that $r|p$ and $1 \ll r \ll p$. Then, [22] observed that for $\mu \in \mathbb{Z}_p$ the following holds. Given $y = (p/r) \cdot \mu + e \mod p$ for small error e, and random additive secret shares $y_0 - y_1 = y \mod p$, it holds

$$\lceil y_0 \rfloor_{p \to r} - \lceil y_1 \rfloor_{p \to r} = \mu \mod r,$$

except with negligible probability. Further, if $|\mu| \ll r$, then this secret sharing holds with overwhelming probability *over the integers* and thus also modulo p:

$$\lceil y_0 \rfloor_{p \to r} - \lceil y_1 \rfloor_{p \to r} = \mu \mod p.$$

Towards obtaining a EOH-PRG, our idea is thus to encode a vector $\mathbf{x} \in \{0,1\}^m$ as $(p/r)\cdot\mathbf{x}$, which then allows to remove errors potentially introduced via the AH-PRG using the conversion function $\mathbf{y} \mapsto \lceil \mathbf{y} \rfloor_{p \to r}$. More precisely, we instantiate the EOH-PRG as in Theorem 8.1 and 8.2.

EOH-PRG from Small-Exponent DCR. Next, we outline our EOH-PRG instantiation from a variant of the DCR assumption. Recall that the DCR assumption induces a homomorphic PRG mapping the additive group $(\mathbb{Z}_{\phi(N)}, +)$ to the multiplicative group $(\mathbb{Z}_{N^2}^*)^4$. Namely, assume $\mathbf{g} := (g_0, g_1, g_2, g_3) \in \mathbb{Z}_{N^2}^4$, where each g_i is uniformly sampled from the N-th residue group mod N^2. Then, based on the DCR assumption, $\mathsf{PRG}_{\mathbf{g}} : \mathbb{Z}_{\phi(N^2)} \to (\mathbb{Z}_{N^2}^*)^4, r \mapsto (g_0^r, g_1^r, g_2^r, g_3^r)$ defines a homomorphic PRG [23], for which it holds $G_{\mathbf{g}}(s_0 - s_1) = G_{\mathbf{g}}(s_0)/G_{\mathbf{g}}(s_1)$ for any $s_0, s_1 \in \mathbb{Z}_{\phi(N^2)}$. However, in order to use this recursively in our constructions, we need to be able to recover a homomorphism over $(\mathbb{Z}_{\phi(N^2)}, +)$ (while not revealing $\phi(N^2)$).

To that end, we follow the techniques of [37], who showed that given $z_0 = z_1 \cdot (1+N)^x \mod N^2$ for $x \in \mathbb{Z}_N$, there exists an efficiently computable map $\mathsf{DDLog}\colon \mathbb{Z}_{N^2}^* \to \mathbb{Z}_N$, which satisfies

$$\mathsf{DDLog}(z_0) - \mathsf{DDLog}(z_1) = x \mod N.$$

Further, if $|x| \leq \frac{N}{2^\lambda}$, then $\mathsf{DDLog}(z_0) - \mathsf{DDLog}(z_1) = x$ over $\mathbb{Z}$, and thus in particular it holds

$$\mathsf{DDLog}(z_0) - \mathsf{DDLog}(z_1) = x \mod \phi(N^2),$$

allowing to recover the homomorphism over $(\mathbb{Z}_{\phi(N^2)}, +)$. Note that this allows to generate secret shares of a value x mod $\phi(N^2)$ *without* knowing $\phi(N^2)$, whenever $|x|$ is sufficiently small.

Our idea is thus to build on a small-exponent variant of the DCR assumption which states that $\mathsf{PRG}_{\mathbf{g}}(r)$ remains a PRG restricted to seeds r with $|r| \leq \frac{N}{2^\lambda}$. It is pointed out in [1] that this variant of the DCR assumption is reasonable as long as the domain of the small exponent is still exponentially large. This kind of low exponent assumption dates back to [36].

With this we can state our EOH-PRG from small-exponent DCR as follows. Let B be an integer such that $B \cdot 2^\lambda \leq N$ and $B > 2^\lambda$. Let $m := \ell(1+w)$ (where, again, ℓ, w are determined by the underlying application). Assume the DCR variant assumption holds relative to exponents in B. Then, we can instantiate the EOH-PRG as in Theorem 8.3.

We would like to point out though that to evaluate the PRG, it is not necessary to know $\phi(N^2)$. The computation mod $\phi(N^2)$ or $\phi(N)$ in the exponent is automatic because of the structure of the Paillier group, and to sample from $\mathbb{Z}_{\phi(N^2)}$, we can sample from $\mathbb{Z}_{N^2}$ instead, as the two distributions are statistically close.

3 Preliminaries

In this section, we recall the preliminaries for function secret sharing from [17,32]. For the remaining preliminaries we refer to the full version [21]. Here, we only consider *two-party* function secret sharing as all of our constructions are in the two-party setting.

Definition 3.1 (Function Secret Sharing (FSS)). *A function secret sharing scheme for function a function class $\mathcal{F}$ consists of two PPT algorithms* $(\mathsf{Gen}, \mathsf{Eval})$*:*

- $\mathsf{Gen}(1^\lambda, f)$ *outputs a pair of keys (k_0, k_1) and correction word CW upon the security parameter and $f \in \mathcal{F}$.*
- $\mathsf{Eval}(b, k_b, CW, x)$ *outputs the corresponding share of $f(x)$ upon the party index b, k_b and input x.*

$(\mathsf{Gen}, \mathsf{Eval})$ *is a secure function secret sharing if it satisfies the correctness and security requirements:*

- **Correctness** *For all $f \in \mathcal{F}$ and all $x \in D_f$,*

$$Pr\left[\mathsf{Eval}(0, k_0, CW, x) - \mathsf{Eval}(1, k_1, CW, x) = f(x) : (k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, f)\right] \geq 1 - \mathsf{negl}(\lambda),$$

  where D_f is the domain of f.
- **Security** *Assume party z is corrupted by an adversary $\mathcal{A}$. Consider the following experiment.*
  1. *The adversary $\mathcal{A}$ outputs $(f_0, f_1) \leftarrow \mathcal{A}(1^\lambda, \mathcal{F})$.*
  2. *The challenger samples $b \leftarrow \{0, 1\}$ and computes $(k_0, k_1, CW) \leftarrow \mathsf{Gen}(1^\lambda, f_b)$.*
  3. *The adversary outputs $b' \leftarrow \mathcal{A}(k_z, CW)$.*

  Let $\mathsf{Adv}(1^\lambda, \mathcal{A})$ be the advantages of $\mathcal{A}$ in guessing b', i.e., $\mathsf{Adv}(1^\lambda, \mathcal{A}) := \left|Pr[b = b'] - \frac{1}{2}\right|$. Then $(\mathsf{Gen}, \mathsf{Eval})$ is a secure FSS if $\mathsf{Adv}(1^\lambda, \mathcal{A})$ is negligible for every $b \in \{0, 1\}$ and every PPT adversary $\mathcal{A}$.

We remark that the FSS definition differs from the FSS in [17,32] in a formal sense, since here the correction word is viewed as an independent part whereas in literature the correction word is part of the party's key.

4 FSS with Additional Properties and EOH-PRGs

In this section, we define *shift-invariant* FSS and *symmetric* FSS, which will serve as a basis for our recursive constructions. We further introduce the notion of a PRG with encoded-output homomorphism (EOH-PRG).

Shift-invariance essentially means that the keys remain functional when shifted by an arbitrary shift s. Note that the shift-invariance does not affect the correction word space $\mathcal{CW}$, which is thus listed separately in Definition 4.1. Further, note that all of the FSS constructions in this work satisfy shift-invariance.

Definition 4.1 (Shift-invariant FSS). *Let* $(\mathsf{Gen}, \mathsf{Eval})$ *be an FSS for a function class* $\mathcal{F}$*. Assume the key space* $\mathcal{K}$ *of* $(\mathsf{Gen}, \mathsf{Eval})$ *is a finite abelian group and the correction word space is* $\mathcal{CW}$*. For any* $f \in \mathcal{F}$*, let* D_f *be the domain of* f*. We say* $(\mathsf{Gen}, \mathsf{Eval})$ *is* shift-invariant*, if there exists a negligible function* $\mathsf{negl}\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ *such that for all* $\lambda \in \mathbb{N}$*,* $f \in \mathcal{F}$*,* $x \in D_f$*,* $(k_0, k_1, CW) \leftarrow \mathsf{Gen}(1^\lambda, f)$*, and* $s \leftarrow_R \mathcal{K}$*,*

$$\Pr\left[\mathsf{Eval}(0, k_0 + s, CW, x) - \mathsf{Eval}(1, k_1 + s, CW, x) = f(x)\right] \geq 1 - \mathsf{negl}(\lambda),$$

where the probability is taken over the randomness of Gen *and* s*.*

Next, we introduce the notion of symmetric FSS. Note that the FSS schemes for point functions in [17,19] are also symmetric.

Definition 4.2 (Symmetric FSS). *An FSS is symmetric if for all* $k \in \mathcal{K}$*, for all* $x \in D_f$*,*

$$\mathsf{Eval}(0, k, CW, x) = \mathsf{Eval}(1, k, CW, x).$$

4.1 PRG with Encoded-Output Homomorphism

We now define the notion *PRG with encoded-output homomorphism* (EOH-PRG), which is central to our work. EOH-PRG corresponds to an approximate substitution of ideal homomorphic PRG.

Note that in the following we consider all entities implicitly parametrized by λ (e.g., by a set S we denote an ensemble of sets $S = \{S_\lambda\}_{\lambda \in \mathbb{N}}$). In Sect. 8, we show how to obtain EOH-PRGs from the (ring)-LWE or DCR assumption.

Definition 4.3 (EOH-PRG). *Let* $\mathbb{S}, \mathbb{H}$ *be finite abelian additive groups, and* $\widetilde{\mathbb{H}}$ *a finite abelian group. Let* $H \subset \mathbb{H}$ *and* $S \subset \mathbb{S}$ *be subsets such that* $0 \in S$*. A function* $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ *is a* PRG with encoded-output homomorphism (EOH-PRG) relative to $(S, H, \mathbb{H})$ *if it is a secure PRG relative to* S *and* $\mathbb{S}$*, and there exists a deterministic polynomial-time encoding function* $\mathsf{Encode} : H \to \widetilde{\mathbb{H}}$ *and a* deterministic *polynomial-time conversion function* $\mathsf{Conv} : \widetilde{\mathbb{H}} \to \mathbb{H}$ *such that for all* $m \in H$*, for* $s \leftarrow_R S$ *and*

$$y := \mathsf{PRG}(s) + \mathsf{Encode}(m),$$

$s_0 \leftarrow_R \mathbb{S}$, $y_0 \leftarrow_R \widetilde{\mathbb{H}}$, $s_1 := s_0 - s$, $y_1 := y_0 - y$ it holds that

$$\mathsf{Conv}(y_0 - \mathsf{PRG}(s_0)) - \mathsf{Conv}(y_1 - \mathsf{PRG}(s_1)) = m$$

in $\mathbb{H}$ except with negligible probability over the choice of s_0 and y_0.

Note that for $y_0 = y_1, s_0 = s_1$, we have $\mathsf{Conv}(y_0 - \mathsf{PRG}(s_0)) = \mathsf{Conv}(y_1 - \mathsf{PRG}(s_1))$ as Conv is deterministic.

Since for our instantiations we will typically have to work with a "tag space" $\mathbb{T}$ operating on $\widetilde{\mathbb{H}}$, we will slightly extend the definition of EOH-PRG, and typically refer to the below when we talk about an EOH-PRG.

Definition 4.4 (EOH-PRG with "tag-space" $\mathbb{T}$). *Let $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ be a EOH-PRG relative to $(S, H, \mathbb{H})$. We say that it is an EOH-PRG relative to $(S, \mathbb{T}, H, \mathbb{H})$, if $\mathbb{T}$ is an additive group such that $T := \{0, 1\} \subset \mathbb{T}$ and there exists a (non-trivial)[10] efficiently computable (left) group operation $\cdot\colon \mathbb{T} \times \widetilde{\mathbb{H}} \to \widetilde{\mathbb{H}}$ of $\mathbb{T}$ on $\widetilde{\mathbb{H}}$.*

Definition 4.5 (EOH-PRG with KDM security). *Let $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ be a EOH-PRG relative to $(S, \mathbb{T}, H, \mathbb{H})$. Let Ψ a family of embeddings $\psi\colon S \to H$. We say PRG is* KDM secure relative to Ψ, *if for all $\psi \in \Psi$, $\mathsf{PRG}^{\psi}\colon s \mapsto \mathsf{PRG}(s) + \mathsf{Encode}(\psi(s))$ is a PRG relative to S and $\mathbb{S}$.*

Remark 4.1. Note that the share obtained from Conv for m may be not pseudorandom. In order to ensure that the homomorphic property can be recursively applied, the two parties can use a PRF with shared key to re-randomize the share of m.

Remark 4.2. For the tensor-product FSS, we need $H = (S \times T)^2$ and $\mathbb{H} = (\mathbb{S} \times \mathbb{T})^2$, for out-degree 2 branching programs we need $H = (S \times \{\mathbf{e}_i\}_{i=1}^{w})^2$ and $\mathbb{H} = (\mathbb{S} \times \mathbb{T}^w)^2$ where w is the width of the branching program and $\mathbf{e}_i$ is the i-th standard basis of $\mathbb{T}^w$.

Remark 4.3. We further need that operations over $\mathbb{S}, \mathbb{H}$ and $\mathbb{T}$ are efficiently computable. While this is trivially the case for our instantiation from LWE, this can be sufficiently emulated for our instantiation with DCR (where $\mathbb{S}$ and $\mathbb{T}$ are additive modulo an unknown $\phi(N)$), by building on techniques of [37].

5 Tensor Product FSS for Arbitrary Predicates from EOH-PRGs

In this section, we present our tensor product FSS, which allows to tensor FSS schemes for arbitrary predicates, as long as the second FSS is symmetric and shift-invariant.

[10] I.e., $1 \cdot h \neq 0$ for $h \neq 0$.

Theorem 5.1 (Tensor Product FSS). *Let $n_1, n_2 \in \mathbb{N}$, and $\mathbb{S}$, $\mathbb{T}$ be two finite abelian groups. Let $\mathcal{P}_1$ be a family of predicates mapping $\{0,1\}^{n_1}$ to $\{0,1\}$ and $\mathcal{P}_2$ be a family of predicates mapping $\{0,1\}^{n_2}$ to $\{0,1\}$. Let $\mathcal{F}_{\mathcal{P}_1} : \{0,1\}^{n_1} \to S \times T, \mathcal{F}_{\mathcal{P}_2} : \{0,1\}^{n_2} \to S \times T$ be the function families induced by $\mathcal{P}_1, \mathcal{P}_2$ as*

$$f_{P_1,\beta} \colon \{0,1\}^{n_1} \to S \times T,\ x \mapsto P_1(x) \cdot \beta = \begin{cases} \beta & \textit{if } P_1(x) = 1 \\ 0 & \textit{else} \end{cases},$$

$$f_{P_2,\gamma} \colon \{0,1\}^{n_2} \to S \times T,\ x \mapsto P_2(x) \cdot \gamma = \begin{cases} \gamma & \textit{if } P_2(x) = 1 \\ 0 & \textit{else} \end{cases},$$

respectively, with $P_1 \in \mathcal{P}_1, P_2 \in \mathcal{P}_2$ and $\beta \in S \times \{1\}, \gamma \in S \times \{1\}$.

Assume

1. $\mathsf{PRG} : \mathbb{S} \to \widetilde{\mathbb{H}}$ *is a EOH-PRG relative to $(S, \mathbb{T}, H, \mathbb{H})$ (as in Definition 4.4), where $\mathbb{H} := (\mathbb{S} \times \mathbb{T})^2$ and $H := (S \times \{0,1\})^2$*
2. $\mathsf{FSS}^{\mathcal{F}_{\mathcal{P}_1}}(\mathsf{Gen}^{\mathcal{F}_{\mathcal{P}_1}}, \mathsf{Eval}^{\mathcal{F}_{\mathcal{P}_1}})$ *is an FSS for $\mathcal{F}_{\mathcal{P}_1}$ over key space $\mathcal{K}_1$, correction word space $\mathcal{CW}_1$ with pseudorandom correction words and pseudorandom output shares.*
3. $\mathsf{FSS}^{\mathcal{F}_{\mathcal{P}_2}}(\mathsf{Gen}^{\mathcal{F}_{\mathcal{P}_2}}, \mathsf{Eval}^{\mathcal{F}_{\mathcal{P}_2}})$ *is a symmetric and shift-invariant FSS for $\mathcal{F}_{\mathcal{P}_2}$ over key space $\mathcal{K}_2 := \mathbb{H}$, such that for all $(u_0, u_1) \leftarrow \mathsf{Gen}^{\mathcal{F}_{\mathcal{P}_2}}$ it holds $u_0 - u_1 \in H$, with correction word space $\mathcal{CW}_2$, and with pseudorandom correction words and output shares.*
4. $\mathsf{PRF} \colon \{0,1\}^\lambda \times [N] \to \mathbb{H}$ *is a PRF (for N sufficiently large).*

Then there exists $\mathsf{FSS}^{\otimes}(\mathsf{Gen}^{\otimes}, \mathsf{Eval}^{\otimes})$ for $\mathcal{G} := \mathcal{F}_{\mathcal{P}_1} \otimes \mathcal{F}_{\mathcal{P}_2} = \{g_{P_1,P_2,\gamma} : \{0,1\}^{n_1} \times \{0,1\}^{n_2} \to S \times T\}$ over key space $\mathcal{K}_1$, correction word space $\mathcal{CW}_1 \times \mathcal{CW}_2 \times \widetilde{\mathbb{H}}$, with pseudorandom correction words and pseudorandom output shares, where

$$g_{P_1,P_2,\gamma}(x_1, x_2) := P_1(x_1) \cdot P_2(x_2) \cdot \gamma = \begin{cases} \gamma & \textit{if } P_1(x_1) = 1 \wedge P_2(x_2) = 1 \\ 0 & \textit{else} \end{cases}.$$

In particular, $\mathsf{FSS}^{\otimes}$ is symmetric and shift-invariant if $\mathsf{FSS}^{\mathcal{F}_{\mathcal{P}_1}}$ is symmetric and shift-invariant.

The construction for $(\mathsf{Gen}^{\otimes}, \mathsf{Eval}^{\otimes})$ is shown in Fig. 1. For the proof we refer to the full version [21]. The FSS for bit-fixing predicates from EOH-PRG in the full version can be viewed as the tensor product of FSS for length 1 predicates.

We further explain how to obtain FSS schemes for the for negation and disjunction of predicates in the full version.

Remark 5.1. As instantiations of the EOH-PRG for N-parties seem out of reach with current techniques without relying on (multi-key) FHE, we did not give the details of the N-party tensor product construction. Roughly, the requirement on shift-invariance of the underlying FSS would become that for $\sum_{i \in [N]} s_i = 0$ it holds $\sum_{i \in [N]} \mathsf{Eval}(i, k_i + s_i, CW, x) = f(x)$ with overwhelming probability, and the requirement on symmetry would become that for $\sum_{i \in [N]} k_i = 0$,

Function secret sharing scheme $\mathsf{FSS}^{\otimes} = (\mathsf{Gen}^{\otimes}, \mathsf{Eval}^{\otimes})$ from EOH-PRG:

Parameters: Let $\mathsf{PRG}\colon \mathbb{S} \to \widetilde{\mathbb{H}}$ be a EOH-PRG relative to $(S, \mathbb{T}, H, \mathbb{H})$, where $\mathbb{H} := (\mathbb{S} \times \mathbb{T})^2$ and $H := (S \times \{0,1\})^2$, and corresponding functions $\mathsf{Encode}, \mathsf{Conv}$. Let $\mathsf{FSS}^{\mathcal{F}_{\mathcal{P}_1}}(\mathsf{Gen}^{\mathcal{F}_{\mathcal{P}_1}}, \mathsf{Eval}^{\mathcal{F}_{\mathcal{P}_1}})$ be an FSS for $\mathcal{F}_{\mathcal{P}_1}$ over key space $\mathcal{K}_1$ and correction word space $\mathcal{CW}_1$. Let $\mathsf{FSS}^{\mathcal{F}_{\mathcal{P}_2}}(\mathsf{Gen}^{\mathcal{F}_{\mathcal{P}_2}}, \mathsf{Eval}^{\mathcal{F}_{\mathcal{P}_2}})$ be a symmetric shift-invariant FSS for $\mathcal{F}_{\mathcal{P}_2}$ over key space $\mathcal{K}_2 = \mathbb{H}$ and correction word space $\mathcal{CW}_2$ such that for any two keys u_0, u_1 in the image of $\mathsf{Gen}^{\mathcal{F}_{\mathcal{P}_2}}$ it holds $u_0 - u_1 \in H$. Further let $N \in \mathbb{N}$ (sufficiently large) and $\mathsf{PRF}\colon \{0,1\}^\lambda \times [N] \to \mathbb{H}$ a PRF. We assume that both parties have access to a global key $K \leftarrow_R \{0,1\}^\lambda$ and global state $\mathsf{st} \in [N]$.

```
Gen^⊗(1^λ, g_{P_1,P_2,γ}) :
  1: Sample $s \leftarrow_R S$ and let $\beta := (s,1)$. Then $\beta \in S \times T \subset \mathbb{S} \times \mathbb{T}$. ▷ $\gamma =: (\sigma, 1) \in S \times T$.
  2: Let $(k_0, k_1, CW_1) \leftarrow \mathsf{Gen}^{\mathcal{F}_{\mathcal{P}_1}}(1^\lambda, f_{P_1,\beta})$.
  3: Let $(u_0, u_1, CW_2) \leftarrow \mathsf{Gen}^{\mathcal{F}_{\mathcal{P}_2}}(1^\lambda, f_{P_2,\gamma})$.    ▷ $u_0, u_1 \in \mathbb{H}$ s.t. $u_0 - u_1 \in H$.
  4: $CW \leftarrow \mathsf{PRG}(s) + \mathsf{Encode}(u_0 - u_1)$.
  5: Let $CW^{\otimes} := (CW_1, CW_2, CW)$ be the new correction word.
  6: Return $(k_0, k_1, CW^{\otimes})$.
Eval^⊗(b, k_b, CW, (x_1, x_2)) :
  1: Parse $CW^{\otimes}$ as $CW^{\otimes} =: (CW_1, CW_2, CW)$.
  2: Let $(s_b, t_b) = \mathsf{Eval}^{\mathcal{F}_{\mathcal{P}_1}}(b, k_b, CW_1, x_1)$.    ▷ $(s_b, t_b) \in \mathbb{S} \times \mathbb{T}$ and
     $(s_0 - s_1, t_0 - t_1) \in S \times T$.
  3: Compute $v_b \leftarrow t_b \cdot CW - \mathsf{PRG}(s_b)$.
  4: Compute $w_b \leftarrow \mathsf{Conv}(v_b) + \mathsf{PRF}(K, \mathsf{st})$.    ▷ $w_b \in \mathbb{H}$ and $w_0 - w_1 \in H$.
  5: Update the state $\mathsf{st} \leftarrow \mathsf{st} + 1$.
  6: Return $\mathsf{Eval}^{\mathcal{F}_{\mathcal{P}_2}}(b, w_b, CW_2, x_2)$.
```

Fig. 1. FSS $(\mathsf{Gen}^{\otimes}, \mathsf{Eval}^{\otimes})$ for $\mathcal{F}_{\mathcal{P}_1} \times \mathcal{F}_{\mathcal{P}_2}$ from $\mathsf{FSS}^{\mathcal{F}_{\mathcal{P}_1}}$, $\mathsf{FSS}^{\mathcal{F}_{\mathcal{P}_2}}$ and EOH-PRG.

it holds $\sum_{i\in[N]} \mathsf{Eval}(i, k_i, CW, x) = 0$. To achieve these properties, the N-party EOH-PRG has to satisfy that (i) $\sum_{i\in[N]} \mathsf{Conv}(c_i - \mathsf{PRG}(s_i)) = m$ for $\sum_{i\in[N]} c_i = \mathsf{PRG}(s) + \mathsf{Encode}(m)$, as well as (ii) $\mathsf{PRG}(0) = 0$ and $\mathsf{Encode}(0) = 0$. Here, requirement (i) is necessary to achieve shift invariance, and the additional requirement (ii) is necessary to achieve symmetry (which is satisfied in both our LWE or DCR instantiation in the two-party case).

6 FSS for Branching Programs

In this section, we generalize the FSS for tensor products to FSS for branching programs. Concretely, the one-bit tag is extended to a w-bit tag, which supports polynomially many possible choices (corresponding to the number of nodes in one level of the branching program). We also generalize the FSS for branching program to FSS for DFAs, approximate matching functions and multivariate polynomials. For details, we refer to Sect. 7 and the full version [21].

Recall that given a branching program P, the size is the number of nodes in V, the length is ℓ, and the width is the maximal number of nodes of every level. Note that every branching program can be converted to a layered, *input-oblivious* branching program with polynomial blowup in size [18,38].

Now, we start to construct an FSS for branching programs. Let P be a layered, oblivious branching program of width w, and let $P_i : \{0,1\}^n \to [w_i]$ be the function which evaluates P to level i (i.e., to the state of the branching program at level i). We start by explaining how to obtain the FSS for the "first level" function

$$f_{P_1,\gamma=(\gamma_0,\gamma_1)} \colon \{0,1\}^n \to (S \times T^{w_1})^2, \mathbf{x} \mapsto \gamma_{x_1},$$

Note that there is one node in level 0 (the initial node) and two nodes in level 1 (one for the choice of 0 and 1 for the choice of 1), i.e., $w_1 = 2$ and P_1 considers only the first bit x_1 of the input $\mathbf{x} \in \{0,1\}^n$.

In order to be able to recurse, we set $\gamma_b := (s_b, \mathbf{t}_b)$, where $s_b \in S$ is some random seed and $\mathbf{t}_0 = (1,0)$ and $\mathbf{t}_1 = (0,1)$ are the unit vectors over $\{0,1\}^2$. A function secret sharing scheme for $f_{P_1,\gamma=(\gamma_0,\gamma_1)}$ which satisfies shift-invariance over $(\mathbb{S}, \mathbb{T}^{w_1})^2$ for some abelian groups $\mathbb{S}, \mathbb{T}$ with $S \subset \mathbb{S}$, $T \subset \mathbb{T}$ can be obtained via a direct truth table sharing.

Lemma 6.1 (Base case). *Let $\mathbb{S}$ and $\mathbb{T}$ be finite abelian groups, and let $S \subset \mathbb{S}$, $T \subset \mathbb{T}$ be arbitrary subsets.*

Then, there exists a shift-invariant FSS for the family of functions $f_{P_1,\gamma}$ over key space $(\mathbb{S} \times \mathbb{T}^2)^2$.

Next, we show an inductive lemma to construct an FSS for P which extends an FSS for P_i to an FSS for P_{i+1}.

Lemma 6.2 (Inductive). *Assume*

1. $\mathsf{FSS}^i = (\mathsf{Gen}^i, \mathsf{Eval}^i)$ *is a shift-invariant FSS for P_i over key space $(\mathbb{S} \times \mathbb{T}^2)^2$, correction word space $\mathcal{CW}_i$ with pseudorandom correction word and pseudorandom output share.* FSS^i *maps the input $\mathbf{x}$ with index set $\{\tau(V_0), \tau(V_1)\ldots, \tau(V_{i-1})\}$ to the $P_i(\mathbf{x})$-th position of a given array β.*
2. $\mathsf{PRG}_i : \mathbb{S} \to \widetilde{\mathbb{H}}_i$ *is a EOH-PRG relative to $(S, \mathbb{T}, H_i, \mathbb{H}_i)$ as in Definition 4.4 where $H_i = (S \times T^{w_{i+1}})^2, \mathbb{H}_i = (\mathbb{S} \times \mathbb{T}^{w_{i+1}})^2$.*

Then, there exists a shift-invariant FSS for P_{i+1} over key space $(\mathbb{S} \times \mathbb{T}^2)^2$, correction word space $\mathcal{CW}_i \times \widetilde{\mathbb{H}}_i^{w_i}$ with pseudorandom correction word and pseudorandom output share. Again, FSS^{i+1} *maps the input $\mathbf{x}$ with index set $\{\tau(V_0), \tau(V_1)\ldots\tau(V_{i-1}), \tau(V_i)\}$ to the $P_{i+1}(\mathbf{x})$-th position of a given array γ.*

The FSS $\mathsf{FSS}^{i+1} = (\mathsf{Gen}^{i+1}, \mathsf{Eval}^{i+1})$ for P_{i+1} is shown in Fig. 2. For the proof of this lemma we refer to the full version [21].

With this, we can obtain an FSS for branching programs of arbitrarily polynomially-bounded width and length, as captured in the following theorem.

Theorem 6.1 (FSS for BP from EOH-PRG). *Let P be a branching program with width $(w_0, w_1, \ldots, w_\ell)$ for each level, where $w_0 = 1, w_1 = 2, w_\ell = 2, w_i \leq w$ for $i \in [0, \ell]$. Assume* $\mathsf{PRG}_i : \mathbb{S} \to \widetilde{\mathbb{H}}_i$ *is a EOH-PRG relative to $S, \mathbb{T}, H_i = (S \times T^{w_{i+1}})^2, \mathbb{H}_i = (\mathbb{S} \times \mathbb{T}^{w_{i+1}})^2$ for $i \in [1, \ell]$.*

Function secret sharing scheme $\mathsf{FSS}^{i+1} = (\mathsf{Gen}^{i+1}, \mathsf{Eval}^{i+1})$ from EOH-PRG for P_{i+1}:

Parameters: Let $\mathsf{PRG}_i : \mathbb{S} \to \widetilde{\mathbb{H}}_i$ be a EOH-PRG relative to $(S, \mathbb{T}, (S \times T^{w_{i+1}})^2, (\mathbb{S} \times \mathbb{T}^{w_{i+1}})^2)$. Let $\mathsf{FSS}^i = (\mathsf{Gen}^i, \mathsf{Eval}^i)$ be a shift-invariant FSS for P_i over key space $\mathcal{K}$ and correction word space $\mathcal{CW}$. Further let $N \in \mathbb{N}$ (sufficiently large) and $\mathsf{PRF}\colon \{0,1\}^\lambda \times [N] \to \mathbb{H}$ is a PRF. We assume that both parties have access to a global key $K \leftarrow_R \{0,1\}^\lambda$ and global state $\mathsf{st} \in [N]$.

```
Gen^{i+1}$(1^\lambda, P_{i+1}, \gamma \in (S \times T^{w_{i+1}})^{w_{i+1}})$ :
 1: Parse $\gamma$ as $\gamma =: ((\sigma[1], \mathbf{e}_1), (\sigma[2], \mathbf{e}_2) \dots (\sigma[w_{i+1}], \mathbf{e}_{w_{i+1}}))$ with $\sigma \in S^{w_{i+1}}$ and $\mathbf{e}_j \in T^{w_{i+1}}$ the $j$-th basis in $\mathbb{T}^{w_{i+1}}$.   ▷ $\sigma$ is the seed value for level $i+1$.
 2: Sample $\mathbf{s} \leftarrow_R S^{w_i}$.   ▷ $\mathbf{s}$ is the seed value for level $i$.
 3: Let $\beta \leftarrow ((\mathbf{s}[1], \mathbf{e}_1), (\mathbf{s}[2], \mathbf{e}_2) \dots (\mathbf{s}[w_i], \mathbf{e}_{w_i})) \in (S \times T^{w_i})^{w_i}$ with $\mathbf{e}_j \in T^{w_i}$ the $j$-th basis in $\mathbb{T}^{w_i}$.
 4: Let $(k_0, k_1, CW_i) \leftarrow \mathsf{Gen}^i(1^\lambda, P_{i,\beta})$.
 5: Let $CW[1 : w_i] \in \widetilde{\mathbb{H}}_i^{w_i}$ be the correction word for level $i$ computed as follows.
 6: for $j \in [w_i]$ do
 7:     Let $\mathbf{u}_j \in (S \times T^{w_{i+1}})^2$ be the key for level $i+1$ with 2 elements.
 8:     Set $\mathbf{u}_j[0] \leftarrow \gamma[f(i,j,0)]$ and $\mathbf{u}_j[1] \leftarrow \gamma[f(i,j,1)]$.   ▷ Map 0 and 1 to the corresponding element of level $i+1$ in the array $\gamma$.
 9:     Set $CW[j] \leftarrow \mathsf{PRG}(\mathbf{s}[j]) + \mathsf{Encode}(\mathbf{u}_j)$.
10: end for
11: Let $CW_{i+1} := (CW_i, CW[1 : w_i])$ be the new correction word.
12: Return $(k_0, k_1, CW_{i+1})$.
Eval^{i+1}$(b, k_b, CW_{i+1}, \mathbf{x})$ :
 1: Parse $CW_{i+1}$ as $CW_{i+1} =: (CW_i, CW[1 : w_i])$.
 2: Let $(s_b, \mathbf{t}_b) = \mathsf{Eval}^i(b, k_b, CW_i, \mathbf{x})$.   ▷ $(s_b, \mathbf{t}_b) \in \mathbb{S} \times \mathbb{T}^{w_i}$.
 3: Compute $v_b \leftarrow \mathsf{Conv}(\sum_{j\in[w]} \mathbf{t}_b[j] \cdot CW[j] - \mathsf{PRG}(s_b)) + \mathsf{PRF}(K, \mathsf{st})$.   ▷ $v_b \in (\mathbb{S} \times \mathbb{T}^{w_{i+1}})^2$.
 4: Update the state $\mathsf{st} \leftarrow \mathsf{st} + 1$.
 5: Return $v_b[\mathbf{x}[\tau(V_{i+1})]] \in \mathbb{S} \times \mathbb{T}^{w_{i+1}}$.   ▷ Use the input $\mathbf{x}[\tau(V_{i+1})]$ to choose the key for level $i+1$.
```

Fig. 2. FSS $(\mathsf{Gen}^{i+1}, \mathsf{Eval}^{i+1})$ for P_{i+1} from FSS $(\mathsf{Gen}^i, \mathsf{Eval}^i)$ and EOH-PRG PRG.

Then, there exists an FSS for P over key space $(\mathbb{S} \times \mathbb{T}^2)^2$ and correction word space $\widetilde{\mathbb{H}}_1^{w_1} \times \widetilde{\mathbb{H}}_2^{w_2} \cdots \times \widetilde{\mathbb{H}}_{\ell-1}^{w_{\ell-1}}$, i.e., with key size bounded by $2(\log|\mathbb{S}| + 2\log|\mathbb{T}|) + \sum_{i\in[1,\ell-1]} w_i \log\left|\widetilde{\mathbb{H}}_i\right|$.

Note that the FSS construction for branching program in Theorem 6.1 only hides the transition function f whereas the topology of the branching program, i.e., the number of nodes of each level, is revealed. For a topology-hiding construction we refer to the full version [21].

7 FSS for DFAs

Similar to the FSS for branching programs, the FSS for DFAs mainly hides the transition function for each state. Given a DFA $M := (Q, \Sigma, \delta, F, q_0)$, the set of accepts states F could be transferred to one accept state A via appending an empty string ϵ to the input whereas the set of other states is transferred to one rejection state R. The transformation leads to a DFA with $|Q|+2$ states and with alphabet $\Sigma \cup \{\epsilon\}$. The construction relies on a KDM secure EOH-PRG.

Theorem 7.1 (FSS for DFAs). *Let $M := (Q, \Sigma, \delta, q_0, F)$ be a DFA. Let $\mu := |Q \cup \{A, R\}| = |Q| + 2$. Assume $\mathsf{PRG} : \mathbb{S} \to \widetilde{\mathbb{H}}$ be a KDM secure EOH-PRG relative to $(S, \mathbb{T}, (S \times T^\mu)^{|\Sigma|+1}, (\mathbb{S} \times \mathbb{T}^\mu)^{|\Sigma|+1})$.*

There exists a FSS for M over key space $\mathbb{S} \times \mathbb{T}^\mu$ and correction word space $\widetilde{\mathbb{H}}^{|Q|}$. Futhermore, the key size is bounded by $\log|\mathbb{S}| + \mu \log|\mathbb{T}| + |Q| \log\left|\widetilde{\mathbb{H}}\right|$.

The FSS for M is shown in Fig. 3.

Function secret sharing scheme for DFA from KDM secure EOH-PRG

Parameters: Let $\mathsf{PRG} : \mathbb{S} \to \widetilde{\mathbb{H}}$ be a KDM secure EOH-PRG relative to $(S, \mathbb{T}, (S \times T^\mu)^{|\Sigma|+1}, (\mathbb{S} \times \mathbb{T}^\mu)^{|\Sigma|+1})$ and $\mu := |Q \cup \{A, R\}| = |Q|+2$. Further let $N \in \mathbb{N}$ (sufficiently large) and $\mathsf{PRF} \colon \{0,1\}^\lambda \times [N] \to \mathbb{H}$ is a PRF. We assume that both parties have access to a global key $K \leftarrow_R \{0,1\}^\lambda$ and global state $\mathsf{st} \in [N]$.

```
Gen($1^\lambda, M$) :
 1: Assign a fixed order to $Q \cup \{A, R\}$ and $\Sigma \cup \{\epsilon\}$.
 2: Sample $\sigma \leftarrow_R S^\mu$.
 3: Let $\beta \leftarrow ((\sigma[1], \mathbf{e}_1) \dots (\sigma[\mu], \mathbf{e}_\mu)) \in (S \times T^\mu)^\mu$ with $\mathbf{e}_j \in T^\mu$ the $j$-th basis in $\mathbb{T}^\mu$.
 4: Let $CW[1 : |Q|]$ be the correction words for meaningful states computed as follows.
 5: for each state $s \in Q$ do
 6:     Let $\mathbf{u} \in (S \times T^\mu)^{|\Sigma|+1}$ be the keys for one-step reachable states from $s$.
 7:     for symbol $\alpha \in \Sigma \cup \{\epsilon\}$ do
 8:         Let $t \leftarrow \delta(s, \alpha)$.
 9:         Set $\mathbf{u}[\alpha] \leftarrow \beta[t]$.                    ▷ $\beta[t]$ is the key for $t$.
10:     end for
11:     Set $CW[s] \leftarrow \mathsf{PRG}(\sigma[s]) + \mathsf{Encode}(\mathbf{u})$.      ▷ No correction words for $\{A, R\}$.
12: end for
13: Sample $k_0, k_1 \leftarrow_R \mathbb{S} \times \mathbb{T}^\mu$ such that $k_0 - k_1 = \beta[q_0]$.      ▷ Share the key for $q_0$.
14: Return $(k_0, k_1, CW)$.
Eval($b, k_b, CW, \mathbf{x}, i$) :
 1: Return $k_b$ if $i > \mathsf{len}(\mathbf{x})$.
 2: Parse $k_b$ as $k_b =: (s_b, \mathbf{t}_b) \in \mathbb{S} \times \mathbb{T}^\mu$.
 3: Compute $v_b \leftarrow \mathsf{Conv}(\sum_{j \in [\mu]} \mathbf{t}_b[j] \cdot CW[j] - \mathsf{PRG}(s_b)) + \mathsf{PRG}(K, \mathsf{st}) \in (\mathbb{S} \times \mathbb{T}^\mu)^{|\Sigma|+1}$.
 4: Update the state $\mathsf{st} \leftarrow \mathsf{st} + 1$.
 5: Return $\mathsf{Eval}(b, v_b[\mathbf{x}[i]], CW, \mathbf{x}, i+1)$.   ▷ Use the input $\mathbf{x}[i]$ to choose the key for
    next state.
```

Fig. 3. FSS for a DFA from KDM secure EOH-PRG.

The correctness is easy to verify as the tag vector is used in Lemma 6.2. The security follows from the pseudorandomness of the KDM secure PRG. We explain why KDM secure EOH-PRG is required. For a state s, it is possible that $\delta(s, \alpha) = s$ for some input $\alpha \in \Sigma$, which leads to the correction word $CW[s] \leftarrow \mathsf{PRG}(\sigma[s]) + \mathsf{Encode}(\mathbf{u})$ and $\sigma[s]$ is also contained in the α-th entry of $\mathbf{u}$. The KDM secure EOH-PRG can be instantiated from LWE or DCR for free without assuming circular security (Remark 8.1). The running time of Eval relies on the input length as DFAs.

Remark 7.1. The FSS for DFAs could be viewed as one level of the FSS for branching programs and there is no level change during the evaluation.

8 EOH-PRG Instantiated from LWE or DCR Assumption

In this section, we show EOH-PRG instantiated from LWE or a DCR variant assumption. As remarked following Definition 4.4, if a PRG is homomorphic then the PRG itself is a EOH-PRG. The LWE assumption implies an almost homomorphic PRG and the DCR assumption implies a homomorphic PRG mapping an additive group to a multiplicative group. We show how the AH-PRG from LWE or the H-PRG from DCR cooperate with other tools to implement the EOH-PRG.

Here we show the three instantiations from LWE, Ring-LWR and DCR. The preliminaries for the assumption, proof of the instantiations and remarks appear in the full version [21].

Theorem 8.1 (EOH-PRG from LWE). *Let $n = n(\lambda), p = p(\lambda), q = q(\lambda), r = r(\lambda), B = B(\lambda) \in \mathbb{N}$ such that $r|p, p|q, 2\lambda^{\omega(1)} \leq r$, $2Br\lambda^{\omega(1)} \leq p$ and $n \log q \leq \ell(n+w) \log p$. Let w, ℓ be parameters depending on concrete applications.*

Assume $\mathsf{LWE}_{n,\ell(n+w),q}$ is hard. Let $S = \{0,1\}^n, T = \{0,1\}, \mathbb{S} = \mathbb{Z}_p^n, \mathbb{T} = \mathbb{Z}_p, H = (S \times T^w)^\ell = \{0,1\}^{\ell(n+w)}, \mathbb{H} = (\mathbb{S} \times \mathbb{T}^w)^\ell = \mathbb{Z}_p^{\ell(n+w)}, \widetilde{\mathbb{H}} = \mathbb{Z}_p^{\ell(n+w)}$ and the corresponding functions for the instantiation be

- *The homomorphic group operation $\cdot : \mathbb{Z}_p \times \mathbb{Z}_p^{\ell(n+w)} \to \mathbb{Z}_p^{\ell(n+w)}, (t, \mathbf{s}) \mapsto t \cdot \mathbf{s}$.*
- PRG: $\mathbb{Z}_p^n \to \mathbb{Z}_p^{\ell(n+w)}, \mathbf{s} \mapsto \lceil \mathbf{s}A \rfloor_{q \to p}$, *where $A \in \mathbb{Z}_q^{n \times \ell(n+w)}$ and the vector-matrix multiplication $\mathbf{s}A$ is performed modulo q;*
- Encode: $\{0,1\}^{\ell(n+w)} \to \mathbb{Z}_p^{\ell(n+w)}, \mathbf{s} \mapsto \frac{p}{r} \cdot \mathbf{s}$;
- Conv: $\mathbb{Z}_p^{\ell(n+w)} \to \mathbb{Z}_p^{\ell(n+w)}, \mathbf{t} \mapsto \lceil \mathbf{t} \rfloor_{p \to r}$.

Then PRG is a EOH-PRG relative to $(S, \mathbb{T}, H, \mathbb{H})$.

Similarly, we show the EOH-PRG instantiation from Ring-LWE. As pointed out in the full version [21], to work with a binary secret Module-LWE instances, the rank of the Module-LWE should be at least $\log q$ (basing on the Ring-LWE pseudorandomness). However, small secret Module-LWR has been used in the NIST post-quantum cryptography submissions including Saber [26], Kyber [41] for constant rank. Based on this, we show the instantiation for good efficiency relying on the Ring-LWE assumption.

Note it is possible that the number $\ell(n+w)$ is not exactly a multiple of n. Assume $\ell \cdot (n+w) = n \cdot \mu + \gamma$ with $0 \leq \gamma < n$. The EOH-PRG from Ring-LWE output has μ elements from $\mathcal{R}_p$ and γ elements from $\mathbb{Z}_p$.

Theorem 8.2 (EOH-PRG from Ring-LWR). *Let $n = n(\lambda), p = p(\lambda), q = q(\lambda), r = r(\lambda), B = B(\lambda) \in \mathbb{N}$ such that $r|p, p|q, 2\lambda^{\omega(1)} \leq r$, $2Br\lambda^{\omega(1)} \leq p$ and $n \log q \leq \ell(n+w)\log p$. Let w, ℓ be parameters depending on concrete applications. Denote $\mathcal{R}$ as the algebraic ring with degree n.*

Assume binary secret $\mathsf{Ring\text{-}LWR}_{\mathcal{R},\ell+\lceil\frac{\ell w}{n}\rceil,q,p}$ is hard. Let $S = \{0,1\}^n, T = \{0,1\}, \mathbb{S} = \mathcal{R}_p, \mathbb{T} = \mathbb{Z}_p, H = (S \times T^w)^\ell = \{0,1\}^{\ell(n+w)}, \mathbb{H} = (\mathbb{S} \times \mathbb{T}^w)^\ell = \mathcal{R}_p^{\ell+\lfloor\frac{\ell w}{n}\rfloor} \times \mathbb{Z}_p^{\ell w - n\lfloor\frac{\ell w}{n}\rfloor}, \widetilde{\mathbb{H}} = \mathcal{R}_p^{\ell+\lfloor\frac{\ell w}{n}\rfloor} \times \mathbb{Z}_p^{\ell w - n\lfloor\frac{\ell w}{n}\rfloor}$ and the corresponding functions for the instantiation be

- *The homomorphic group operation $\cdot\colon \mathbb{Z}_p \times \left(\mathcal{R}_p^\ell \times \mathbb{Z}_p^{\ell w - n\lfloor\frac{\ell w}{n}\rfloor}\right) \to \mathcal{R}_p^\ell \times \mathbb{Z}_p^{\ell w - n\lfloor\frac{\ell w}{n}\rfloor}, (t,s) \mapsto t \cdot s$, where $\cdot$ is the scalar multiplication mod p.*
- *PRG: $\mathcal{R}_p \to \mathcal{R}_p^\ell \times \mathbb{Z}_p^{\ell w - n\lfloor\frac{\ell w}{n}\rfloor}, s \mapsto \psi(\lceil s \cdot \mathbf{a}\rfloor_{q\to p})$, where $\mathbf{a} \in \mathcal{R}_q^{\ell+\lceil\frac{\ell w}{n}\rceil}$, the multiplication $s \cdot \mathbf{a}$ is performed modulo $\mathcal{R}_q$, and $\psi(\cdot)$ takes the ring elements except the last one and the first $\ell w - n\left\lceil\frac{\ell w}{n}\right\rfloor$ coefficients of the last ring element;*
- *Encode: $\{0,1\}^{\ell(n+w)} \to \mathcal{R}_p^\ell \times \mathbb{Z}_p^{\ell w - n\lfloor\frac{\ell w}{n}\rfloor}, \mathbf{s} \mapsto \frac{p}{r} \cdot s$;*
- *Conv: $\mathcal{R}_p^\ell \times \mathbb{Z}_p^{\ell w - n\lfloor\frac{\ell w}{n}\rfloor} \to \mathcal{R}_p^\ell \times \mathbb{Z}_p^{\ell w - n\lfloor\frac{\ell w}{n}\rfloor}, t \mapsto \lceil t\rfloor_{p\to r}$.*

Then PRG is a EOH-PRG relative to $(S, \mathbb{T}, H, \mathbb{H})$.

Next we show how the EOH-PRG is instantiated from the DCR variant assumption It is pointed out in [1, Section 4.1], the DCR variant assumption is sound if the domain of the small exponent is exponentially large. This kind of low exponent assumption dates back to [36] and was also used in [16]. To enable the homomorphic group operation, here we use $\mathbb{Z}_{\phi(N^2)}$ instead of $\mathbb{Z}_{\phi(N)}$ for the additive group.

Theorem 8.3 (EOH-PRG from DCR). *Let B be an integer such that $B \cdot 2^\lambda \leq N$ and $B > 2^\lambda$.*

Assume the DCR variant assumption holds. Let $S = [-\frac{B}{2}, \frac{B}{2}], T = \{0,1\}, \mathbb{S} = \mathbb{Z}_{\phi(N^2)}, \mathbb{T} = \mathbb{Z}_{\phi(N^2)}, H = (S \times T^w)^\ell, \mathbb{H} = (\mathbb{S} \times \mathbb{T}^w)^\ell = \mathbb{Z}_{\phi(N^2)}^{\ell(1+w)}, \widetilde{\mathbb{H}} = (\mathbb{Z}_{N^2}^*)^{\ell(1+w)}$ and the corresponding functions for the instantiation be

- *The homomorphic group operation $\cdot\colon \mathbb{Z}_{\phi(N^2)} \times (\mathbb{Z}_{N^2}^*)^{\ell(1+w)} \to (\mathbb{Z}_{N^2}^*)^{\ell(1+w)}$, $(t, \mathbf{s}) \mapsto \mathbf{s}^t \mod N^2$.*
- *PRG : $\mathbb{Z}_{\phi(N^2)} \to (\mathbb{Z}_{N^2}^*)^{\ell(1+w)}, s \mapsto \mathbf{g}^s$, where $\mathbf{g} \in (\mathbb{Z}_{N^2}^*)^{\ell(1+w)}$ and each entry of $\mathbf{g}$ is a N-th residue;*
- *Encode : $([-B/2, B/2] \times \{0,1\}^w)^\ell \to (\mathbb{Z}_{N^2}^*)^{\ell(1+w)}, \mathbf{m} \mapsto (1+N)^{\mathbf{m}} \mod N^2$;*
- *Conv : $(\mathbb{Z}_{N^2}^*)^{\ell(1+w)} \to \mathbb{Z}_{\phi(N^2)}^{\ell(1+w)}, \mathbf{t} \mapsto \mathsf{DDLog}(\mathbf{t})$.*

Then PRG is a EOH-PRG relative to $(S, \mathbb{T}, H, \mathbb{H})$.

Note that the secret key $\phi(N)$ for Paillier encryption is not *explicitly* used in the operations of instantiation of EOH-PRG from DCR. The computation mod

$\phi(N^2)$ or $\phi(N)$ in the exponent is automatic because of the structure of the Paillier group, and to sample from $\mathbb{Z}_{\phi(N^2)}$, we can sample from $\mathbb{Z}_{N^2}$ instead, as the two distributions are statistically close.

Remark 8.1 (KDM Security). The FSS constructions for DFAs in Sect. 7 rely on the KDM security of EOH-PRG. It is straightforward to prove the pseudorandomness for LWE or DCR following the method to prove the KDM security in [3, Theorem 6] or [9, Section 3.2] as detailed in the full version [21].

Acknowledgements. E. Boyle's research is supported in part by AFOSR Award FA9550-21-1-0046 and ERC Project HSS (852952). L. Kohl is funded by NWO Talent Programme Veni (VI.Veni.222.348) and by NWO Gravitation project QSC. Z. Li is funded by NWO Gravitation project QSC and ERC ADG ALGSTRONGCRYPTO (740972). P. Scholl is funded by the Independent Research Fund Denmark under project number 0165-00107B (C3PO).

References

1. Abram, D., Damgård, I., Orlandi, C., Scholl, P.: An algebraic framework for silent preprocessing with trustless setup and active security. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part IV. LNCS, vol. 13510, pp. 421–452 (Aug 2022)
2. Alamati, N., Montgomery, H., Patranabis, S., Roy, A.: Minicrypt primitives with algebraic structure and applications. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 55–82 (May 2019)
3. Applebaum, B., Cash, D., Peikert, C., Sahai, A.: Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 595–618 (Aug 2009)
4. Barni, M., Failla, P., Kolesnikov, V., Lazzeretti, R., Sadeghi, A.R., Schneider, T.: Secure evaluation of private linear branching programs with medical applications. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 424–439 (Sep 2009)
5. Barni, M., Failla, P., Lazzeretti, R., Sadeghi, A., Schneider, T.: Privacy-preserving ECG classification with branching programs and neural networks. IEEE Trans. Inf. Forensics Secur. 6(2), 452–468 (2011)
6. Benhamouda, F., Degwekar, A., Ishai, Y., Rabin, T.: On the local leakage resilience of linear secret sharing schemes. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 531–561 (Aug 2018)
7. Bogdanov, A., Guo, S., Masny, D., Richelson, S., Rosen, A.: On the hardness of learning with rounding over small modulus. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016-A, Part I. LNCS, vol. 9562, pp. 209–224 (Jan 2016)
8. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Lightweight techniques for private heavy hitters. In: 2021 IEEE Symposium on Security and Privacy. pp. 762–776. IEEE Computer Society Press (May 2021)
9. Boneh, D., Halevi, S., Hamburg, M., Ostrovsky, R.: Circular-secure encryption from decision Diffie-Hellman. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 108–125 (Aug 2008)
10. Boneh, D., Lewi, K., Montgomery, H.W., Raghunathan, A.: Key homomorphic PRFs and their applications. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 410–428 (Aug 2013)

11. Bost, R., Popa, R.A., Tu, S., Goldwasser, S.: Machine learning classification over encrypted data. In: NDSS 2015. The Internet Society (Feb 2015)
12. Boyle, E.: Function Secret Sharing (2022), http://cyber.biu.ac.il/wp-content/uploads/2021/11/FSS-2022-BIU-WinterSchool_Elette.pdf, The 12th BIU Winter School on cryptography - Advances in Secure Computation
13. Boyle, E., Chandran, N., Gilboa, N., Gupta, D., Ishai, Y., Kumar, N., Rathee, M.: Function secret sharing for mixed-mode and fixed-point secure computation. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part II. LNCS, vol. 12697, pp. 871–900 (Oct 2021)
14. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 896–912. ACM Press (Oct 2018)
15. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 489–518 (Aug 2019)
16. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Orrù, M.: Homomorphic secret sharing: Optimizations and applications. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 2105–2122. ACM Press (Oct / Nov 2017)
17. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 337–367 (Apr 2015)
18. Boyle, E., Gilboa, N., Ishai, Y.: Breaking the circuit size barrier for secure computation under DDH. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016, Part I. LNCS, vol. 9814, pp. 509–539 (Aug 2016)
19. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing: Improvements and extensions. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 1292–1303. ACM Press (Oct 2016)
20. Boyle, E., Gilboa, N., Ishai, Y.: Secure computation with preprocessing via function secret sharing. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019, Part I. LNCS, vol. 11891, pp. 341–371 (Dec 2019)
21. Boyle, E., Kohl, L., Li, Z., Scholl, P.: Direct FSS constructions for branching programs and more from PRGs with encoded-output homomorphism. Cryptology ePrint Archive, Report 2024/192 (2024), https://eprint.iacr.org/2024/192
22. Boyle, E., Kohl, L., Scholl, P.: Homomorphic secret sharing from lattices without FHE. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 3–33 (May 2019)
23. Brakerski, Z., Goldwasser, S.: Circular and leakage resilient public-key encryption under subgroup indistinguishability - (or: Quadratic residuosity strikes back). In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 1–20 (Aug 2010)
24. Brickell, J., Porter, D.E., Shmatikov, V., Witchel, E.: Privacy-preserving remote diagnostics. In: Ning, P., De Capitani di Vimercati, S., Syverson, P.F. (eds.) ACM CCS 2007. pp. 498–507. ACM Press (Oct 2007)
25. Corrigan-Gibbs, H., Boneh, D., Mazières, D.: Riposte: An anonymous messaging system handling millions of users. In: 2015 IEEE Symposium on Security and Privacy. pp. 321–338. IEEE Computer Society Press (May 2015)
26. D'Anvers, J.P., Karmakar, A., Roy, S.S., Vercauteren, F., Mera, J.M.B., Beirendonck, M.V., Basso, A.: SABER. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions

27. Dauterman, E., Rathee, M., Popa, R.A., Stoica, I.: Waldo: A private time-series database from function secret sharing. In: SP 2022. pp. 2450–2468. IEEE (2022), https://doi.org/10.1109/SP46214.2022.9833611
28. Davidson, A., Katsumata, S., Nishimaki, R., Yamada, S., Yamakawa, T.: Adaptively secure constrained pseudorandom functions in the standard model. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 559–589 (Aug 2020)
29. Dodis, Y., Halevi, S., Rothblum, R.D., Wichs, D.: Spooky encryption and its applications. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016, Part III. LNCS, vol. 9816, pp. 93–122 (Aug 2016)
30. Doerner, J., shelat, a.: Scaling ORAM for secure computation. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 523–535. ACM Press (Oct / Nov 2017)
31. Fazio, N., Gennaro, R., Jafarikhah, T., Skeith III, W.E.: Homomorphic secret sharing from paillier encryption. In: Okamoto, T., Yu, Y., Au, M.H., Li, Y. (eds.) ProvSec 2017. LNCS, vol. 10592, pp. 381–399 (Oct 2017)
32. Gilboa, N., Ishai, Y.: Compressing cryptographic resources. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 591–608 (Aug 1999)
33. Gilboa, N., Ishai, Y.: Distributed point functions and their applications. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 640–658 (May 2014)
34. Ishai, Y., Paskin, A.: Evaluating branching programs on encrypted data. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 575–594 (Feb 2007)
35. Kiss, Á., Naderpour, M., Liu, J., Asokan, N., Schneider, T.: SoK: Modular and efficient private decision tree evaluation. PoPETs 2019(2), 187–208 (Apr 2019)
36. Koshiba, T., Kurosawa, K.: Short exponent Diffie-Hellman problems. In: Bao, F., Deng, R., Zhou, J. (eds.) PKC 2004. LNCS, vol. 2947, pp. 173–186 (Mar 2004)
37. Orlandi, C., Scholl, P., Yakoubov, S.: The rise of paillier: Homomorphic secret sharing and public-key silent OT. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part I. LNCS, vol. 12696, pp. 678–708 (Oct 2021)
38. Pippenger, N.: On simultaneous resource bounds (preliminary version). In: FOCS. pp. 307–311. IEEE Computer Society (1979)
39. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM journal of research and development 3(2), 114–125 (1959)
40. Roy, L., Singh, J.: Large message homomorphic secret sharing from DCR and applications. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part III. LNCS, vol. 12827, pp. 687–717. Virtual Event (Aug 2021)
41. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Seiler, G., Stehlé, D., Ding, J.: CRYSTALS-KYBER. Tech. rep., National Institute of Standards and Technology (2022), available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
42. Servan-Schreiber, S., Langowski, S., Devadas, S.: Private approximate nearest neighbor search with sublinear communication. In: SP. pp. 911–929. IEEE (2022), https://doi.org/10.1109/SP46214.2022.9833702
43. Sipser, M.: Introduction to the theory of computation. PWS Publishing Company (1997)
44. Tueno, A., Kerschbaum, F., Katzenbeisser, S.: Private evaluation of decision trees using sublinear cost. PoPETs 2019(1), 266–286 (Jan 2019)
45. Wang, F., Yun, C., Goldwasser, S., Vaikuntanathan, V., Zaharia, M.: Splinter: Practical private queries on public data. In: NSDI 2017 (2017), https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wang-frank

46. Wegener, I., Woelfel, P.: New results on the complexity of the middle bit of multiplication. In: CCC. pp. 100–110. IEEE Computer Society (2005)
47. Wu, D.J., Feng, T., Naehrig, M., Lauter, K.E.: Privately evaluating decision trees and random forests. PoPETs 2016(4), 335–355 (Oct 2016)

Dishonest Majority Multiparty Computation over Matrix Rings

Hongqing Liu(✉), Chaoping Xing, Chen Yuan, and Taoxu Zou

Shanghai Jiao Tong University, Shanghai, China
{liu.hong.qing,xingcp,chen_yuan,seasun}@sjtu.edu.cn

Abstract. The privacy-preserving machine learning (PPML) has gained growing importance over the last few years. One of the biggest challenges is to improve the efficiency of PPML so that the communication and computation costs of PPML are affordable for large machine learning models such as deep learning. As we know, linear algebra such as matrix multiplication occupies a significant part of the computation in deep learning such as deep convolutional neural networks (CNN). Thus, it is desirable to propose the MPC protocol specialized for the matrix operations. In this work, we propose a dishonest majority MPC protocol over matrix rings which supports matrix multiplication and addition. Our MPC protocol can be seen as a variant of SPDZ protocol, i.e., the MAC and global key of our protocol are vectors of length m and the secret of our protocol is an $m \times m$ matrix. Compared to the classic SPDZ protocol, our MPC protocol reduces the communication complexity by at least m times to securely compute a matrix multiplication. We also show that the communication complexity of our MPC protocol is asymptotically as good as [16] which also presented a dishonest majority MPC protocol specialized for matrix operations, i.e., the communication complexity of securely computing a multiplication gate is $O(m^2n^2 \log q)$ in the preprocessing phase and $O(m^2 n \log q)$ in the online phase. The share size and the number of multiplications of our protocol are reduced by around 50% and 40% of [16], respectively. However, we take a completely different approach. The protocol in [16] uses a variant of BFV scheme to embed a whole matrix into a single ciphertext and then treats the matrix operation as the entry-wise operation in the ciphertext while our approach resorts to a variant of vector linear oblivious evaluation (VOLE) called the subfield VOLE (In [33], there is a base VOLE which is also called subfield VOLE. The subfield VOLE in this paper is referred to the programmable VOLE $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$ in [33] which silently generates correlated randomness from seeds) [33] which can securely compute the additive sharing of $v\boldsymbol{x}$ for $v \in \mathbb{F}_{q^b}, \boldsymbol{x} \in \mathbb{F}_q^a$ with sublinear communication complexity. Finally, we note that our MPC protocol can be easily extended to small fields.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-981-96-0938-3_10.

© International Association for Cryptologic Research 2025
K.-M. Chung and Y. Sasaki (Eds.): ASIACRYPT 2024, LNCS 15489, pp. 299–327, 2025.
https://doi.org/10.1007/978-981-96-0938-3_10

1 Introduction

Secure multiparty computation (MPC) allows a set of mutually distrustful parties $P_1, \cdots, P_n$ to jointly compute a public function f with their private inputs, and reveals nothing except the final output. The adversary could corrupt at most t of n parties to gain the private information of honest parties by either inspecting the transcripts between parties (semi-honest adversary) or arbitrarily deviating from the protocol (malicious adversary). According to the number of corrupted parties t, MPC protocols can be classified into two categories: honest majority ($t \leq \frac{n}{2}$) and dishonest majority ($t < n$). The honest majority MPC protocol can achieve information-theoretic security while the dishonest majority MPC protocol can only achieve computational security.

In MPC protocols, the public function f is generally modeled as an arithmetic circuit over a finite field or a ring, which consists of addition and multiplication gates. The MPC protocols over a ring are usually more complicated than those over a field. Before the advent of privacy preserving machine learning (PPML), most of the MPC protocols were restricted to the computation over finite fields. The use of integer rings is well-motivated in practice due to their direct compatibility with hardware. In view of this practical application, a line of works [2,3,18,23,31] proposed the MPC protocol over $\mathbb{Z}_{2^k}$. Recently, Escudero and Soria-Vazquez [22] considered the non-commutative ring in the honest majority setting. They constructed an unconditionally secure MPC over non-commutative rings with black-box access to a ring containing an exceptional set[1], whose size is at least the number of parties. They also proposed an honest majority MPC protocol over the matrix ring $\mathcal{M}_{m\times m}(\mathbb{Z}_{2^k})$.

Inspired by [22], a natural question is can we design an MPC protocol over a non-commutative ring with only black-box access to the ring in the presence of $t \geq \frac{n}{2}$ corrupted parties? The answer is probably negative as the dishonest majority MPC protocols rely on some cryptographic assumptions. Moreover, while honest majority MPC protocols use the error-correction algorithm of Shamir secret sharing to detect and even correct the corruptions, the dishonest majority MPC protocols have to rely on the additive secret sharing scheme to protect the privacy of the data which has no room to detect the corruptions. Therefore, message authenticate codes (MACs) are commonly attached to the additive secret sharing scheme to detect the corruptions, which are highly related to the concrete structure of the non-commutative ring.

In view of the above reasons, we aim to construct a dishonest majority MPC over a specific family of the non-commutative ring, the matrix ring. Matrix plays an essential role in PPML, which allows distrustful parties to train and evaluate different machine learning models [25,28–30]. It was observed in [16] that securely multiplying two $m \times m$ matrices in SPDZ protocol requires at least $O(m^{2.8})$ authenticated Beaver triples, which is prohibitively expensive if a machine learning task needs a large number and sizes of matrix multiplication. Thus, an MPC protocol specialized for matrix operations may greatly improve

[1] A subset of a non-commutative ring where the difference between any two elements in this subset is invertible.

the efficiency of PPML. Moreover, some other non-commutative rings could be represented in the form of matrix rings. For instance, the quaternion ring is another non-commutative ring with practical applications, which plays a central role in computer graphics and aerospace due to its competence in describing the rotation in three-dimensional space.

In this work, we present a variant of SPDZ protocol whose secret is defined over matrix rings. Different from the classic SPDZ protocol, the MAC and global key of our protocol are vectors of length m and the secret of our protocol is an $m \times m$ matrix. Thus, the size of our MAC is sublinear in the size of our secret assuming the size of our matrix is large enough. Utilizing the matrix structure, our MPC protocol uses vector oblivious linear evaluation (VOLE) and vector oblivious product evaluation (VOPE) as functionalities to authenticate the sharing and create the sextuple for securely computing multiplication gates in the online phase. The goal of VOPE is to compute the additive sharing of the product of two matrices which can be adapted from the subfield VOLE in [33] with slight modification. In the preprocessing phase, our MPC protocol needs $O(n^2 m^2 \log q)$ bits of communication to prepare a sextuple for multiplication gate which has the same asymptotic performance as the protocol in [16]. In the online phase, our MPC protocol requires $O(m^2 n \log q)$ bits of communication complexity to securely compute a multiplication gate which is also as efficient as the MPC protocol in [16]. However, the size of the secret sharing is half the size of the secret sharing scheme in [16] and the number of multiplications in our protocol is reduced to $3m^3 + 3m^2$ while the protocol in [16] requires $5m^3 + m^2$ multiplication. We also compare the communication cost and computation cost of preprocessing in concrete parameter settings for $m = 128, 256, 512, 1024$. When m grows, the communication complexity of our protocol grows more slowly than [16]. For $m = 512, 1024$, the communication complexity of our protocol turns out to be smaller than [16]. Moreover, our experimental results imply that the running time of our VOLE-based preprocessing phase is 2.0x-24.2x faster than that of (fully) homomorphic encryption based preprocessing phase [16].

1.1 Our Contribution

MAC for Matrix Rings. To authenticate a matrix $M \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$, we choose a uniformly random vector $\boldsymbol{v} \in \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ as the global key and use the matrix-vector product $M\boldsymbol{v}$ as the MAC of a matrix M. The intuition of this matrix-vector product is to reduce the size of MAC by applying the batch check, i.e., each component of the MAC is the inner product of a row of M and the global key $\boldsymbol{v}$. If the adversary aims to forge a fake authenticated secret sharing, he needs to choose a nonzero matrix $E \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and a vector $\boldsymbol{\delta} \in \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ such that $E\boldsymbol{v} = \boldsymbol{\delta}$. Since E is a nonzero matrix, we assume that the i-th row of E is a nonzero vector $\boldsymbol{e}_i^T$. Then, we have $\boldsymbol{e}_i^T \boldsymbol{v} = \delta_i$ where δ_i is the i-th component of $\boldsymbol{\delta}$. Since the global key $\boldsymbol{v}$ is distributed uniformly at random, the adversary succeeds with probability at most $1/q$. In comparison, the previous MPC protocol in [16] chooses a random element $\alpha \in \mathbb{F}_q$ as the global key and uses the scalar-matrix product αM as the corresponding MAC. Therefore, our MAC is m times smaller than theirs. The sharing of the matrix M

in our protocol is defined as $\langle M \rangle = ([M], [\![\boldsymbol{v}]\!], [\![M\boldsymbol{v}]\!])$[2] where $[M]$ is the additive sharings of M and $[\![\boldsymbol{v}]\!]$, $[\![M\boldsymbol{v}]\!]$ are the additive sharing of $\boldsymbol{v}$ and $M\boldsymbol{v}$ respectively.

The Use of VOLE. Our protocol uses the vector oblivious linear evaluation (VOLE for short) to compute the matrix-vector product. We exploit the matrix structure to optimize the generation of correlated randomness. In the computation of MAC, two parties need to obliviously compute the product of a matrix M with a column vector $\boldsymbol{v}$, i.e., $\boldsymbol{u} + \boldsymbol{w} = M\boldsymbol{v}$. Observe that $M\boldsymbol{v}$ can be decomposed into the sum of m vectors $v_i\boldsymbol{m}_i$ where v_i is the i-th component of $\boldsymbol{v}$ and $\boldsymbol{m}_i$ is the i-th column of M. Two parties can invoke VOLE m times to obtain the shares $\boldsymbol{u}_i, \boldsymbol{w}_i$ with $\boldsymbol{u}_i + \boldsymbol{v}_i = v_i\boldsymbol{m}_i$. In contrast, we have to invoke m^2 OLEs to obliviously compute $M\boldsymbol{v}$, which is usually more expensive than VOLE.

The Use of Subfield VOLE. The subfield VOLE was proposed in [12] to securely compute the additive sharings of $v\boldsymbol{x}$ for $\boldsymbol{x} \in \mathbb{F}_q^a, v \in \mathbb{F}_{q^b}$ where v and $\boldsymbol{x}$ are the random element and random vector input by P_A and P_B respectively. To minimize the communication cost, the random vector $\boldsymbol{x}$ is expanded by a random seed while the random element v is chosen by P_B. Treating v as a vector $\boldsymbol{v} \in \mathbb{F}_q^b$, then $v\boldsymbol{x}$ becomes a product of two vectors $\boldsymbol{x}\boldsymbol{v}^T = (x_i v_j)_{a\times b}$. In this sense, the subfield VOLE can securely compute the additive sharing of $\boldsymbol{x}\boldsymbol{v}^T$ for $\boldsymbol{x} \in \mathbb{F}_q^a, \boldsymbol{v} \in \mathbb{F}_q^b$. We slightly modify the subfield VOLE in [33] to allow both parties to utilize short seeds to generate their random inputs. We call this modified subfield VOLE, random vector oblivious product evaluation (VOPE). Assuming $b = O(a)$, our VOPE has $O(a \log q)$ communication complexity, which is sublinear in output size $ab \log q$.

Computing the Product of Matrices. We propose the VOPE to compute the additive sharings of the product of two random matrices whose communication complexity is the dominant part of the preprocessing phase. Observe that one can decompose the product AB of two matrices $A, B \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ into the sum of vector product $\boldsymbol{a}_i \otimes \boldsymbol{b}_i := \boldsymbol{a}_i\boldsymbol{b}_i^T$, $i \in [m]$ where $\boldsymbol{a}_i$ is the i-th column of A and $\boldsymbol{b}_i^T$ is the i-th row of B, i.e., $AB = \sum_{i=1}^m \boldsymbol{a}_i \otimes \boldsymbol{b}_i$. As mentioned above, our VOPE can produce the additive sharings of $\boldsymbol{a}_i \otimes \boldsymbol{b}_i$. Thus, it suffices to invoke m times of VOPE to obtain the additive sharing of AB.

Multiplication Sextuple. The biggest challenge of MPC protocol over matrix rings is that the product of two matrices is not commutative. This prevents us from applying the Beaver triple straightforwardly. This problem also appears in [23]. Their solution is to use two types of secret sharings with left linearity and right linearity respectively and transform the type of secret sharing by consuming a double sharing, which is a pair of sharings associated with the same secret and different types. In our case, since our MAC has the form $X\boldsymbol{v}$, our secret sharing

[2] We use $[\cdot]$ and , $[\![\cdot]\!]$ to represent the sharing of a matrix and vector, respectively.

only allows left multiplication, i.e., all parties can only locally compute $A\langle M\rangle = \langle AM\rangle$. We propose a multiplication sextuple to circumvent this obstacle. Let $\langle X\rangle$ and $\langle Y\rangle$ be the sharings of matrix X and Y respectively. We prepare a sextuple $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$ where A, B, R are random matrices, A^T and R^T are the transpose of A and R, and $C = AB$. All parties partially open $\langle X\rangle - \langle A\rangle$ and $\langle Y\rangle - \langle B\rangle$ to D and E. The technique of Beaver triple requires all parties to locally compute $D\langle B\rangle + \langle A\rangle E + \langle C\rangle + DE$. However, as we mentioned above, it is impossible to locally compute the right multiplication $\langle A\rangle E$. To overcome this obstacle, all parties are required to locally compute $E^T\langle A^T\rangle - \langle R^T\rangle$ and partially open it to F by using the sharing $\langle A^T\rangle$ and $\langle R^T\rangle$. Then, all parties locally compute $F^T + \langle R\rangle = \langle AE\rangle$ by observing $F^T = (E^T A^T - R^T)^T = AE - R$. This completes the multiplication gate.

Function-Dependent Preprocessing. The evaluation of a single multiplication gate in our MPC protocol needs two rounds and three broadcasts. Inspired by [9,21], we introduce function-dependent preprocessing to improve the round and communication complexity. After the application of function-dependent preprocessing, the evaluation of a multiplication gate only needs one round and two broadcasts. Since this improvement is not the focus of our paper, we take a brief overview of it in the full version [27].

Migration to small field $\mathbb{F}_q$. The matrix in our MPC protocol can be defined over small fields as well. The idea is to replace a global key of a vector in $\mathcal{M}_{m\times 1}(\mathbb{F}_q)$ with a global key of a matrix in $\mathcal{M}_{m\times \ell}(\mathbb{F}_q)$. The intuition is that the adversary succeeds with probability $1/q$ if our MPC protocol is defined over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$. To reduce the error probability, we increase the size of the global key and MAC. Observe that $XV = \varDelta$ where $V \in \mathcal{M}_{m\times \ell}(\mathbb{F}_q)$ is the MAC and $X \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ is the secret. Therefore, each column of the global key is used to verify the correctness of the secret and we verify our secret X with ℓ equations instead of 1. The error probability will be reduced to $1/q^\ell$ while the size of MAC is still sublinear in the size of our secret assuming $m \gg \frac{\kappa}{\log_2 q}$. In this sense, our MPC protocol can be defined over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ with any prime power q. There are also some modifications for our MPC protocol to be applicable to $\mathcal{M}_{m\times m}(\mathbb{F}_q)$. The details can be found in the full version [27].

1.2 Overview of Our Technique

We assume that our MPC protocol over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ with large q. As we have mentioned above, the authenticated sharing of our protocol is $\langle M\rangle = ([M], [\![\boldsymbol{v}]\!], [\![M\boldsymbol{v}]\!])$. We use a random vector $\boldsymbol{v}$ as our global key. The MAC of our matrix is the product of a matrix with the global key $\boldsymbol{v}$. The idea of our MAC comes from the batch check. A random vector can be used to verify the correctness of a vector of the same length by taking the inner product of these two vectors. Thus, to verify the correctness of an $m \times m$ matrix, we only need a MAC of size m. On the contrary, the classic SPDZ protocol requires MAC of size m^2 to verify an $m \times m$ matrix. Another merit of this sharing can be found in the use of VOLE and VOPE which we have already discussed in Sect. 1.1.

In the preprocessing phase, our MPC protocol prepares sextuples of the form $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$ with random matrices $A, B, R \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and $C = AB$. We break this protocol into two procedures, π_{Mult} and π_{Double}. We also present a protocol Π_{Auth} to generate the authenticated sharing. Protocol Π_{Auth} uses functionality VOLE to create the MAC and takes the random linear combination to verify the correctness of sharings.

Procedure π_{Mult} produces a triple $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$. We want to compute $[C]$ from $[A] = \left(A^{(1)}, \dots, A^{(n)}\right)$ and $[B] = \left(B^{(1)}, \dots, B^{(n)}\right)$. Observe that $C = AB = \left(\sum_{i=1}^{n} A^{(i)}\right)\left(\sum_{i=1}^{n} B^{(i)}\right)$. The additive sharing of cross terms $A^{(i)}B^{(j)}$ and $A^{(j)}B^{(i)}$ can be computed by P_i and P_j. The product of two $m\times m$ matrices can be decomposed into the sum of m vector products as we mentioned above, i.e., $AB = \sum_{i=1}^{m} \boldsymbol{a}_i \otimes \boldsymbol{b}_i$ where $\boldsymbol{a}_i$ is the i-th column of A and $\boldsymbol{b}_i^T$ is the i-th row of B. This implies that we only need to invoke m times VOPE to complete this work. We create seeds to generate the random matrix $A^{(i)}$, $B^{(i)}$ and reuse these seeds as inputs for the instances of VOPE. The use of VOPE can be found in the previous section. We fix B and apply the above process twice to $([A], [B])$ and $([A'], [B])$ to prepare two pairs $([A], [C]), ([A'], [C'])$ with $C = AB, C' = A'B$. Then, we invoke protocol Π_{Auth} to compute the MAC of these sharings. By taking a random linear combination of the form $\chi\langle A\rangle - \langle A'\rangle$, we can verify the product relation and output the authenticated triple $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$.

Procedure π_{Double} takes inputs $\langle A_i\rangle, i \in [2\ell]$ and outputs pairs of authenticated sharing $\langle A_i\rangle, \langle A_i^T\rangle, i \in [2\ell]$ for ℓ multiplication gates. The idea is to first locally compute $[A_i^T]$ from $[A_i]$ by applying the transpose to each share in $[A_i]$. Then, we apply protocol Π_{Auth} to create the authenticated sharing $\langle A_i^T\rangle$. To check the transpose relation, we generate a pair of authenticated sharing of random matrix A_0, A_0^T and sacrifice this pair by taking the random linear combination

$$\langle C\rangle = \sum_{i=1}^{2\ell} r_i \langle A_i\rangle + \langle A_0\rangle \quad \langle D\rangle = \sum_{i=1}^{2\ell} r_i \langle A_i^T\rangle + \langle A_0^T\rangle$$

It must hold that $C = D^T$. Then, this procedure will output pairs of authenticated sharing $\langle A_i\rangle, \langle A_i^T\rangle, i \in [2\ell]$.

In the online phase, our MPC protocol can securely compute the addition and multiplication gate. The addition gate can be locally computed without interaction. To compute the multiplication gate, we need a sextuple prepared in the preprocessing phase. This sextuple can help us to circumvent the obstacle that the product of two matrices is non-commutative. One can find the details in Sect. 1.1.

1.3 Related Work

There are a few MPC protocols optimized for matrix operations. Escudero and Soria-Vazquez [22] presented an honest majority MPC protocol over matrix rings. One of the biggest challenges in their protocol is to construct Shamir secret sharing scheme over non-commutative rings. They constructed a subset of matrices as the evaluation points such that these matrices are commutative. Based on this subset of matrices, they presented the encoding and error correction algorithm for this Shamir secret sharing scheme. Since our MPC protocol is secure in the presence of dishonest majority, our building block is an additive secret sharing scheme. The sharing and reconstruction algorithm can be straightforwardly generalized from the commutative case. However, we need a MAC to verify the correctness of our sharing whose idea can be dated back to SPDZ protocol [20]. In our protocol, the global key and the MAC are vectors instead of elements. Thus, the MAC of our protocol is negligible compared to the size of the secret.

The most relevant work is due to [16] which presented a variant of SPDZ protocol over matrix rings $\mathcal{M}_{m\times m}(\mathbb{Z}_q)$, where $\mathbb{Z}_q$ is a large prime field. They mimic the classic SPDZ protocol to use a single element as the global key to create the MAC of the matrix. Thus, the size of MAC in their protocol is as big as the secret. In the preprocessing phase, they apply the homomorphic matrix multiplication [25] which is based on a variant of BFV scheme [14,24] to create the matrix triple. Their SPDZ protocol over matrix rings turns out to be very efficient compared to the classic SPDZ protocol handling the matrix operations as the entry-wise operations.

In the preprocessing phase, we apply a variant of PCG-based subfield VOLE to securely multiply two random matrices. In [13], Boyle et al. proposed a PCG construction for matrix triple, which is adapted from the PCG for OLE under "splittable" ring-LPN assumption. However, their protocol generates a large batch of matrix triples of small-to-medium size (at most 16×16), while our protocol can deal with the matrices of large size (at least 128×128).[3]

1.4 Organization of the Paper

The paper is organized as follows. In Sect. 2, we present basic notations and definitions. In Sect. 3, we present the online phase of our MPC protocol. In Sect. 4, we present Protocol Π_{Auth} which outputs authenticated sharings. In Sect. 5, we present the preprocessing phase of our MPC protocol. In Sect. 6, we analyze the communication complexity of our MPC protocol and compare it with other dishonest majority MPC protocols over matrix rings. The missing functionalities and protocols can be found in Section A in the Supplementary Material.

2 Preliminaries

2.1 Basic Notation

We use the capital letter M to represent a matrix and bold small letter $\boldsymbol{v}$ to represent a column vector. The transpose of a matrix M is M^T and the transpose

[3] In [13], they remarked "For larger matrix, more interactive approach such as the recent work based on homomorphic encryption [16] appears to be more practical".

of a vector $\boldsymbol{v}$ is $\boldsymbol{v}^T$. For a vector $\boldsymbol{v}$, denote by v_i the i-th component of $\boldsymbol{v}$, i.e., $\boldsymbol{v}^T = (v_1, \ldots, v_n)$. Let $\mathcal{M}_{a\times b}(\mathbb{F}_q)$ be the collection of $a \times b$ matrices over $\mathbb{F}_q$. For two column vectors $\boldsymbol{u} \in \mathbb{F}_q^a, \boldsymbol{v} \in \mathbb{F}_q^b$, we use $\boldsymbol{u} \otimes \boldsymbol{v} = \boldsymbol{u}\boldsymbol{v}^T \in \mathcal{M}_{a\times b}(\mathbb{F}_q)$ to represent their (outer) product, i.e., $\boldsymbol{u}\boldsymbol{v}^T = (u_i v_j)_{a\times b}$ where $\boldsymbol{u}^T = (u_1, \ldots, u_a)$ and $\boldsymbol{v}^T = (v_1, \ldots, v_b)$.

Throughout the paper, the security parameter of MPC protocol is κ. Let $\mathbb{F}_q$ be the finite field of size q and $\mathbb{F}_q^n$ be the vector space of n dimension. We denote by $x \overset{\$}{\leftarrow} \mathcal{X}$ a variable x uniformly sampling from a finite set $\mathcal{X}$. Let $[N] = \{1, \cdots, N\}$.

2.2 Multiparty Computation

The set of parties in our MPC protocol is $\{P_1, \cdots, P_n\}$. We consider the setting of dishonest majority, where at most $n-1$ parties are corrupted by the adversary. The adversary is static and malicious, which means that the set of corrupted parties is determined before the execution of protocol and corrupted parties can arbitrarily deviate from the protocol.

The security of our protocol is proved under Canetti's Universal Composability (UC) framework [15]. A protocol Π securely instantiates a functionality $\mathcal{F}$ if there exists a simulator that interacts with the adversary (or more formally, *environment*) such that he can distinguish the ideal world and real world with only negligible probability. The composability of UC framework enables us to construct our protocol in *hybrid* model, which means that protocol Π instantiates functionality $\mathcal{F}$ with access to another functionality $\mathcal{F}'$. In this case, Π instantiates $\mathcal{F}$ in the $\mathcal{F}'$-hybrid model. Different from a protocol Π which is associated with an ideal functionality and has simulation-based proof, we use π to represent a procedure, which acts as a subroutine of protocols, and has no related functionality or simulation-based proof.

We assume the private and authenticated channels between any pair of parties and a broadcast channel. Our MPC protocol achieves security with (unanimous) abort since the majority of parties are dishonest. In the ideal world, the functionality waits for a signal from the adversary before delivery of outputs. If the signal is Abort, all honest parties abort. Otherwise, the signal is OK, the functionality sends correct outputs to all honest parties. In the real world, when we say a party aborts, this party sends an Abort signal through the broadcast channel and all honest parties abort.

3 Online Phase

We begin by introducing the authenticated secret sharing of a matrix, which is the building block of our MPC protocol. Protocol Π_{online} securely implements MPC functionality $\mathcal{F}_{\mathsf{MPC}}$ in the $(\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}_{\mathsf{Coin}})$-hybrid model, where $\mathcal{F}_{\mathsf{Prep}}$ generates correlated randomness in offline phase and $\mathcal{F}_{\mathsf{Coin}}$ generates public random field elements. The implementation of $\mathcal{F}_{\mathsf{Prep}}$ can be found in Sect. 5.

3.1 Authenticated Secret Sharing

In the dishonest majority setting, additive secret sharing alone is not resilient to the corruption caused by the malicious adversary. Similar to [19], we use a uniformly random global key to generate the MAC of the share. Our approach deviates from [19] by making the global key and MACs as a vector of length m over $\mathbb{F}_q$.

Notations. We use $[\cdot]$ and $[\![\cdot]\!]$ to denote an additive secret sharing over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ and $\mathcal{M}_{m\times 1}(\mathbb{F}_q)$[4], respectively. An authenticated secret sharing $\langle X\rangle$ is a triple $([X],[\![\boldsymbol{v}]\!],[\![X\boldsymbol{v}]\!])$, where $X \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$ is the secret, $\boldsymbol{v} \xleftarrow{\$} \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ is the global key and $X\boldsymbol{v} \in \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ is the MAC of the secret. More precisely, $[X] = \left(X^{(1)},\cdots,X^{(n)}\right)$, $[\![\boldsymbol{v}]\!] = \left(\boldsymbol{v}^{(1)},\cdots,\boldsymbol{v}^{(n)}\right)$ and $([\![X\boldsymbol{v}]\!]) = \left(\boldsymbol{m}^{(1)}(X),\cdots,\boldsymbol{m}^{(n)}(X)\right)$ with

$$X = \sum_{i=1}^{n} X^{(i)}, \boldsymbol{v} = \sum_{i=1}^{n} \boldsymbol{v}^{(i)}, X\boldsymbol{v} = \sum_{i=1}^{n} \boldsymbol{m}^{(i)}(X).$$

where party P_i holds random share $X^{(i)}$ of secret X, key share $\boldsymbol{v}^{(i)}$ and MAC share $\boldsymbol{m}^{(i)}(X)$.

Local Operations. We use "linear" to refer to "$\mathcal{M}_{m\times m}(\mathbb{F}_q)$-linear". Scheme $[\cdot]$ is both *left linear* and *right linear* due to distribute law of matrix rings. However, scheme $\langle\cdot\rangle$ is only *left linear*. Given an authenticated secret sharing $\langle X\rangle$ and a public matrix $A \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$, all parties could left multiply A to $[\![X\boldsymbol{v}]\!]$ to obtain $[\![AX\boldsymbol{v}]\!]$, but it is not possible to obtain $[\![XA\boldsymbol{v}]\!]$ with only local operations. To securely left multiply a matrix A with $\langle X\rangle$, all parties locally compute

$$A\langle X\rangle = \langle AX\rangle = ([AX],[\![\boldsymbol{v}]\!],[\![AX\boldsymbol{v}]\!])$$

with $[AX] = (AX^{(1)},\ldots,AX^{(n)})$ and $[\![AX\boldsymbol{v}]\!] = (A\boldsymbol{m}^{(1)}(X),\ldots,A\boldsymbol{m}^{(n)}(X))$. To securely compute the sum of $\langle X\rangle$ and $\langle Y\rangle$, all parties locally compute

$$\langle X\rangle + \langle Y\rangle = \langle X+Y\rangle = ([X+Y],[\![\boldsymbol{v}]\!],[\![(X+Y)\boldsymbol{v}]\!])$$

with $[X+Y] = (X^{(1)}+Y^{(1)},\ldots,X^{(n)}+Y^{(n)})$ and $[\![(X+Y)\boldsymbol{v}]\!] = \big(\boldsymbol{m}^{(1)}(X) + \boldsymbol{m}^{(i)}(Y),\ldots,\boldsymbol{m}^{(n)}(X)+\boldsymbol{m}^{(n)}(Y)\big)$. To securely add a public matrix A with $\langle X\rangle$, all parties locally compute

$$[X+A] = (X^{(1)}+A, X^{(2)},\ldots,X^{(n)}), \boldsymbol{m}^{(i)}(X+A) = \boldsymbol{m}^{(i)}(X) + A\boldsymbol{v}^{(i)}$$

Then, $\langle X+A\rangle = ([X+A],[\![\boldsymbol{v}]\!],[\![(X+A)\boldsymbol{v}]\!])$ is the authenticated secret sharing of $X+A$. The affine operation can be found in procedure π_{Aff} in Section A in the Supplementary Material.

[4] Here we use notion $\mathcal{M}_{m\times 1}(\mathbb{F}_q)$ instead of $\mathbb{F}_q^m$ in order to show that the global key and MACs can be generalized to matrix.

Opening and Checking. To partially open an authenticate secret sharing $\langle Y\rangle = ([Y], [\![\boldsymbol{v}]\!], [\![Y\boldsymbol{v}]\!])$, all parties send their shares of $[Y]$ to P_1, who can reconstruct the secret and send the result Y' to other parties. To verify the opened value Y', all parties locally compute $[\![\boldsymbol{\sigma}]\!] = [\![Y\boldsymbol{v}]\!] - Y'[\![\boldsymbol{v}]\!]$, and broadcast the shares of this value via a simultaneous message channel. The parties abort if the reconstructed value σ is not $\mathbf{0}$. The probability that a fake authenticated secret sharing passes the verification is $1/q$. These two procedures can be found in Section A in the Supplementary Material.

Multiplication. In dishonest majority MPC protocols, correlated randomness generated in offline phase could assist the computation of multiplications. Beaver triple [8] is a common technique in MPC protocols, which transforms execution of multiplications to broadcasts and linear operations. However, we can not adapt Beaver triple directly due to the non-commutative property of matrix ring.

To multiply two authenticated sharings $\langle X\rangle$ and $\langle Y\rangle$, all parties prepare a Beaver triple $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$ with $C = AB$ during the preprocessing phase. All parties partially open $D \leftarrow \langle X\rangle - \langle A\rangle$ and $E \leftarrow \langle Y\rangle - \langle B\rangle$. The sharing of $Z = XY$ could be represented as:

$$[Z] = [C] + D[B] + [A]E + DE$$
$$[\![Z\boldsymbol{v}]\!] = [\![C\boldsymbol{v}]\!] + D[\![B\boldsymbol{v}]\!] + [\![AE\boldsymbol{v}]\!] + DE[\![\boldsymbol{v}]\!]$$

We observe that all items except $[\![AE\boldsymbol{v}]\!]$ could be locally computed with linear operations. To compute MAC share $[\![AEv]\!]$, we follow the paradigm of "mask-open-unmask". We choose a random sharing $[R]$ as the mask of $[A]E$. However, when opening the masked value $[A]E - [R]$, we cannot guarantee the correctness due to the lack of MAC. Therefore, we prepare two additional authenticated sharings $(\langle A^T\rangle, \langle R^T\rangle)$ and partially open the transpose $\langle F\rangle = E^T\langle A^T\rangle - \langle R^T\rangle$ instead. Therefore, to execute a multiplication, all parties need to prepare a *multiplication sextuple* $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$ where $A, B, R \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and $C = AB$.

3.2 Required Functionalities

The functionality $\mathcal{F}_{\mathsf{MPC}}$ enables the parties to securely share their inputs, perform linear operations and multiplications, and output the result. The functionality $\mathcal{F}_{\mathsf{Prep}}$ is used to prepare correlated randomness for $\mathcal{F}_{\mathsf{MPC}}$.

Authenticating functionality $\mathcal{F}_{\mathsf{Auth}}$. This functionality allows parties to generate the shares of global key $\boldsymbol{v}$ and transform an additive secret sharing $[X]$ to an authenticated secret sharing $\langle X\rangle$. Although we do not call $\mathcal{F}_{\mathsf{Auth}}$ directly, $\mathcal{F}_{\mathsf{Auth}}$ is contained in $\mathcal{F}_{\mathsf{Prep}}$.

Functionality 1: $\mathcal{F}_{\mathsf{Auth}}$

Let $\mathcal{C}$ be the set of corrupted parties.

- **Initialize**: On receiving (Init) from all parties, sample random vector $\boldsymbol{v}^{(i)} \leftarrow \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ for $i \notin \mathcal{C}$ and receive $\boldsymbol{v}^{(i)}$ from adversary for $i \in \mathcal{C}$. Store the global key $\boldsymbol{v} = \sum_{i=1}^{n} \boldsymbol{v}^{(i)}$ and send $\boldsymbol{v}^{(i)}$ to P_i.
- **Authenticate**: On receiving $(\mathsf{Auth}, [X])$ from each party P_i, where $[X]$ is an additive sharing over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$:
  1. Compute the MAC $\boldsymbol{m}(X) = X\boldsymbol{v}$.
  2. Wait for $\left\{\boldsymbol{m}^{(i)}(X)\right\}_{i\in\mathcal{C}}$ from adversary and sample $\left\{\boldsymbol{m}^{(i)}(X)\right\}_{i\notin\mathcal{C}}$ subject to $\sum_{i=1}^{n} \boldsymbol{m}^{(i)}(X) = \boldsymbol{m}(X)$.
  3. $\mathcal{S}$ sends $\boldsymbol{m}^{(j)}(X)$ to P_j for all $j \notin \mathcal{C}$.

Preprocessing Functionality $\mathcal{F}_{\mathsf{Prep}}$. This functionality produces random sharings for input gates and multiplication sextuples for multiplication gates.

Functionality 2: $\mathcal{F}_{\mathsf{Prep}}$

The functionality has all the same commands in $\mathcal{F}_{\mathsf{Auth}}$, with following additional commands:

- **Input**: On input $(\mathsf{InputPrep}, P_i)$ from all parties, sample $R \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and generate its authenticated sharing $\langle R\rangle$ such that for $j \in \mathcal{C}$, $\left(R^{(j)}, \boldsymbol{m}^{(j)}(R)\right)$ is chosen by the adversary. Output R to P_i and $\left(R^{(j)}, \boldsymbol{m}^{(j)}(R)\right)$ to P_j for all $j \notin \mathcal{C} \cup \{i\}$.
- **Sextuple**: On input (Tuple) from all parties, sample $A, B, R \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$ and compute $C = AB$. Generate authenticated sharings $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$ such that for $j \in \mathcal{C}$, j-th shares of these sharings are chosen by the adversary.

Multiparty Computation Functionality $\mathcal{F}_{\mathsf{MPC}}$. This functionality provides all the necessary operations for our MPC protocol.

Functionality 3: $\mathcal{F}_{\mathsf{MPC}}$

The functionality maintains a dictionary Val, which keeps a track of authenticated elements in $\mathcal{M}_{m\times m}(\mathbb{F}_q)$. For each authenticated secret sharing, the shares of corrupted parties can be chosen by the adversary.

- **Initialize**: On input (Init) from all parties, set the global key $[\![\boldsymbol{v}]\!]$.
- **Input**: On input $(\mathsf{Input}, \mathtt{id}, X, P_i)$ from P_i and $(\mathsf{Input}, \mathtt{id}, P_i)$ from all other parties, store $\mathsf{Val}[\mathtt{id}] = X$.

- **Addition**: On input $(\mathsf{Add}, \mathtt{id}, \mathtt{id}_1, \mathtt{id}_2)$ from all parties, compute $Z = \mathsf{Val}[\mathtt{id}_1] + \mathsf{Val}[\mathtt{id}_2]$ and store $\mathsf{Val}[\mathtt{id}] = Z$.
- **Public matrix multiplication**: On input $(\mathsf{PubMul}, \mathtt{id}, A)$, compute $Z = A\mathsf{Val}[\mathtt{id}]$ and store $\mathsf{Val}[\mathtt{id}] = Z$.
- **Multiplication**: On input $(\mathsf{Mult}, \mathtt{id}, \mathtt{id}_1, \mathtt{id}_2)$ from all parties, compute $Z = \mathsf{Val}[\mathtt{id}_1]\mathsf{Val}[\mathtt{id}_2]$ and store $\mathsf{Val}[\mathtt{id}] = Z$.
- **Check openings**: On input $(\mathsf{Check}, (\mathtt{id}_1, \cdots, \mathtt{id}_\ell), (X'_1, \cdots, X'_\ell))$ from all parties, wait for a signal for the adversary. If the adversary sends OK and $\mathsf{Val}[\mathtt{id}_j] = X'_j$ for $j \in [\ell]$, return OK to all honest parties. Otherwise, return Abort to all honest parties.
- **Output**: On input $(\mathsf{Output}, \mathtt{id})$ from all parties, the functionality retrieves $Y = \mathsf{Val}[\mathtt{id}]$ and sends Y to the adversary if $\mathsf{Val}[\mathtt{id}] \neq \emptyset$. If the adversary sends Abort then the functionality aborts, otherwise it delivers Y to all parties.

Coin tossing functionality $\mathcal{F}_{\mathsf{Coin}}$. This functionality generates a uniformly random element in $\mathbb{F}_q$ for all parties.

Functionality 4: $\mathcal{F}_{\mathsf{Coin}}$

Upon receiving (Coin) from all parties, sample $r \xleftarrow{\$} \mathbb{F}_q$ and send r to the adversary.

- If the adversary returns OK, send r to all honest parties.
- If the adversary returns Abort, send Abort to all honest parties.

3.3 Instantiation of $\mathcal{F}_{\mathsf{MPC}}$

The protocol Π_{online} instantiates $\mathcal{F}_{\mathsf{MPC}}$ in the $(\mathcal{F}_{\mathsf{Prep}},\mathcal{F}_{\mathsf{Coin}})$-hybrid model, with statistical security parameter κ. The random shares and multiplication sextuples produced in $\mathcal{F}_{\mathsf{Prep}}$ will be used in Input and Mult commands, respectively.

Protocol 1: Π_{Online}

The parties maintain a dictionary Val for authenticated values.

- **Initialize**: The parties call $\mathcal{F}_{\mathsf{Prep}}$ as follows:
  1. On input (Init) to get global key $[\![\boldsymbol{v}]\!]$.
  2. On input $(\mathsf{InputPrep}, P_i)$ to prepare a random authenticated sharing $\langle R \rangle$ for each input gate, where the input provider P_i learns R.
  3. On input (Tuple) to prepare a multiplication sextuple $(\langle A \rangle, \langle A^T \rangle, \langle B \rangle, \langle C \rangle, \langle R \rangle, \langle R^T \rangle)$ for each multiplication gate
- **Input**: If P_i receives $(\mathsf{Input}, \mathtt{id}, X, P_i)$ and other parties receive $(\mathsf{Input}, \mathtt{id}, P_i)$, execute following operations:
  1. P_i broadcasts $A = X - R$, where $\langle R \rangle$ is an unused input mask

2. All parties locally compute $\langle X\rangle = \langle R\rangle + A$ and store $\mathsf{Val}[\mathtt{id}] = \langle X\rangle$.

- **Addition**: If all parties receive $(\mathsf{Add}, \mathtt{id}, \mathtt{id}_1, \mathtt{id}_2)$, retrieve $\langle X\rangle = \mathsf{Val}[\mathtt{id}_1]$ and $\langle Y\rangle = \mathsf{Val}[\mathtt{id}_2]$, locally compute $\langle Z\rangle = \langle X\rangle + \langle Y\rangle$ and set $\mathsf{Val}[\mathtt{id}] = \langle Z\rangle$.
- **Public matrix multiplication**: If all parties receive $(\mathsf{PubMul}, \mathtt{id}, A)$, retrieve $\langle X\rangle = \mathsf{Val}[\mathtt{id}]$, locally compute $\langle Z\rangle = A\langle X\rangle$ and set $\mathsf{Val}[\mathtt{id}] = \langle Z\rangle$.
- **Multiplication**: If all parties receive $(\mathsf{Mult}, \mathtt{id}, \mathtt{id}_1, \mathtt{id}_2)$, retrieve $\langle X\rangle = \mathsf{Val}[\mathtt{id}_1]$ and $\langle Y\rangle = \mathsf{Val}[\mathtt{id}_2]$ and execute following operations:
  1. Choose an unused multiplication sextuple $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$.
  2. All parties locally compute $\langle D\rangle \leftarrow \langle X\rangle - \langle A\rangle$ and $\langle E\rangle \leftarrow \langle Y\rangle - \langle B\rangle$.
  3. All parties partially open $D \leftarrow \pi_{\mathsf{Open}}(\langle D\rangle)$ and $E \leftarrow \pi_{\mathsf{Open}}(\langle E\rangle)$.
  4. All parties locally compute $\langle F\rangle \leftarrow E^T\langle A^T\rangle - \langle R^T\rangle$ and partially open $F \leftarrow \pi_{\mathsf{Open}}(\langle F\rangle)$
  5. All parties locally compute $\langle Z\rangle = \langle C\rangle + D\langle B\rangle + \langle R\rangle + DE + F^T$ and set $\mathsf{Val}[\mathtt{id}] = \langle Z\rangle$.
- **Check openings**: If all parties receive $(\mathsf{Check}, (\mathtt{id}_1, \cdots, \mathtt{id}_\ell), (X'_1, \cdots, X'_\ell))$, retrieve $\langle X_j\rangle = \mathsf{Val}[\mathtt{id}_j]$ for $j \in [\ell]$ and execute following operations:
  1. Call $\mathcal{F}_{\mathsf{Coin}}$ ℓ times to sample $r_1, \cdots, r_\ell \overset{\$}{\leftarrow} \mathbb{F}_q$.
  2. All parties locally compute $\langle Y\rangle \leftarrow \sum_{j=1}^{\ell} r_j\langle X_j\rangle$.
  3. All parties locally compute $Y' = \sum_{j=1}^{\ell} r_j X'_j$.
  4. All parties invoke $\pi_{\mathsf{Check}}(Y', \langle Y\rangle)$.
- **Output:** If all parties receive $(\mathsf{Output}, \mathtt{id})$ and retrieve $\langle Y\rangle = \mathsf{Val}[\mathtt{id}]$:
  1. All parties invoke Check command to check all the opened values in the online phase so far.
  2. If this does not abort, the parties partially open $\langle Y\rangle$ to obtain Y'.
  3. All parties invoke $\pi_{\mathsf{Check}}(Y', \langle Y\rangle)$. If this procedure passes, output Y'.

Theorem 1. *Protocol Π_{Online} securely implements $\mathcal{F}_{\mathsf{MPC}}$ in the $(\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}_{\mathsf{Coin}})$-hybrid model.*

Proof. A full-fledged simulation-based proof is presented in the full version [27]. Here we restrict ourselves to the core idea of the proof. For the case of Init command, it is easy to see that the shares of the global key are prepared for all parties on both Π_{Online} and $\mathcal{F}_{\mathsf{MPC}}$. In the Input command, the value stored by $\mathcal{F}_{\mathsf{MPC}}$ corresponds to the value stored by Π_{Online}, which can be seen authenticated through the mask of a random share.

The case of Add and PubMul is easy since these steps only consist of local computations which can be simulated trivially. To analyze Mult command, we should take three values into consideration. The correctness of the multiplication step in $\mathcal{F}_{\mathsf{MPC}}$ is easy to be verified. The parties obtain a tuple $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$ before computing the product Z of two stored values $X, Y \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$. The parties first partially opens $D \leftarrow X - A$ and $E \leftarrow Y - B$, and then compute locally $[Z] = [C] + D[B] + [A]E + DE$, which is equivalent to $[Z] = [XY]$. The third value $F^T \leftarrow E^T A^T - R^T$ is opened to compute the MAC of Z. The parties can locally compute $[\![Z\boldsymbol{v}]\!] =$

$[\![C\boldsymbol{v}]\!] + D[\![B\boldsymbol{v}]\!] + F[\![\boldsymbol{v}]\!] + [\![R\boldsymbol{v}]\!] + DE[\![\boldsymbol{v}]\!]$. We can verify that the formula is equivalent to $[\![Z\boldsymbol{v}]\!] = [\![XY\boldsymbol{v}]\!]$. Note that each time we partially open a value, we compute its MAC. This MAC will be used in the Check command to check the correctness of this opened value. The privacy argument is clear as we always mask our secret with a random matrix when we want to do partially opening.

Finally, in the Check and Output command, we can prove that a corrupted authenticated secret sharing will pass the verification with a negligible probability due to the following game which first appears in [20].

1. The challenger generates the secret key $\boldsymbol{v} \xleftarrow{\$} \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ and MACs $\boldsymbol{\gamma}_i = X_i\boldsymbol{v}$ for $i \in [\ell]$ and sends $X_1, \ldots, X_\ell$ to the adversary.
2. The adversary sends back $X'_1, \ldots, X'_\ell$.
3. The challenger generates the random values $r_1, \ldots, r_\ell \in \mathbb{F}_q$.
4. The adversary provides an error $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_m)^T$.
5. The adversary checks that $\{\sum_{i=1}^{\ell} r_i(X_i - X'_i)\}\boldsymbol{v} = \boldsymbol{\delta}$

The adversary wins the game if the check passes and exists $X_i - X'_i \neq 0$. The second step of the game reveals that corrupted parties have the option to lie about the secret shares that they opened during the execution of the protocol. $\boldsymbol{\delta}$ models the fact that the adversary is allowed to introduce errors on the MAC. Suppose $\sum_{i=1}^{\ell} r_i(X_i - X'_i)$ is not an all-zero matrix and let the nonzero row be $(x_{a,1}, \ldots, x_{a,m})$. We have $\delta_a = \sum_{j=1}^{m} x_{a,j}v_j$. Since $\boldsymbol{v} = (v_1, \ldots, v_m)^T$ is kept secret from the adversary, the adversary wins the game with the probability at most $1/q$. Now we proceed to the case $\sum_{i=1}^{\ell} r_i(X_i - X'_i) = 0$. Because $r_1, \ldots, r_\ell$ are random elements, the probability that $\sum_{i=1}^{\ell} r_iE_i = 0$ for not all-zero matrix E_i is at most $1/q$. Thus, the adversary wins this game with probability at most $1/q$.

4 Authentication

In this section, we show how to authenticate an additive secret sharing. We first introduce a cryptographic primitive VOLE and then show how to generate the MAC share by invoking the VOLE.

4.1 Required Functionalities

Vector Oblivious Linear Evaluation Functionality $\mathcal{F}_{\mathsf{VOLE}}$. A VOLE is a two-party functionality between P_A and P_B, which takes as input a vector $\boldsymbol{x}$ from the sender P_A and a scalar v from the receiver P_B, then randomly samples a vector $\boldsymbol{w}$ and computes $\boldsymbol{u} = v\boldsymbol{x} + \boldsymbol{w}$. We need to invoke VOLE multiple times and thus we attach a unique identifier *sid* to each instance[5]. The efficient instantiation of $\mathcal{F}_{\mathsf{VOLE}}$ can be found in [4,5].

[5] The unique identifier *sid* is locally shared among a pair of parties and thus is not a global identifier in n-party setting.

Functionality 5: $\mathcal{F}^{sid}_{\mathsf{VOLE}}$

The functionality runs between sender P_A and receiver P_B. The **Initialize** step runs once at the beginning and the **Multiply** step could run multiple times.

– **Initialize**: Upon receiving $v \in \mathbb{F}_q$ from P_B, store v.
– **Multiply**: Upon receiving $\boldsymbol{x} \in \mathbb{F}_q^m$ from P_A:
  1. Sample $\boldsymbol{w} \xleftarrow{\$} \mathbb{F}_q^m$. If P_B is corrupted, receive $\boldsymbol{w}$ from adversary.
  2. Compute $\boldsymbol{u} = v\boldsymbol{x} + \boldsymbol{w}$. If P_A is corrupted, receive $\boldsymbol{u}$ from adversary and recompute $\boldsymbol{w} = \boldsymbol{u} - v\boldsymbol{x}$.
  3. Output $\boldsymbol{u}$ to P_A and $\boldsymbol{w}$ to P_B.

4.2 Instantiation of $\mathcal{F}_{\mathsf{Auth}}$

Now we proceed to the generation of MAC shares. Each party P_i randomly samples the global key share $\boldsymbol{v}^{(i)}$ when command Init is invoked. To authenticate a given share $\{X^{(i)}\}_{i\in[n]}$, all parties jointly compute the additive sharing of $\left(\sum_{i=1}^n X^{(i)}\right)\left(\sum_{i=1}^n \boldsymbol{v}^{(i)}\right)$. Observe that:

$$\left(\sum_{i=1}^n X^{(i)}\right)\left(\sum_{i=1}^n \boldsymbol{v}^{(i)}\right) = \sum_{i=1}^n X^{(i)}\boldsymbol{v}^{(i)} + \sum_{i\neq j} X^{(i)}\boldsymbol{v}^{(j)}$$

Each party P_i can locally compute $X^{(i)}\boldsymbol{v}^{(i)}$ and each ordered pair (P_i, P_j) needs to interactively compute additive sharing of $X^{(i)}\boldsymbol{v}^{(j)}$, i.e., $\boldsymbol{u}^{(i,j)} + \boldsymbol{w}^{(j,i)} = X^{(i)}\boldsymbol{v}^{(j)}$, where P_i and P_j receives $\boldsymbol{u}^{(i,j)}$ and $\boldsymbol{w}^{(j,i)}$, respectively. By setting $\boldsymbol{m}^{(i)}(X) = X^{(i)}\boldsymbol{v}^{(i)} + \sum_{j\neq i}\left(\boldsymbol{u}^{(i,j)} + \boldsymbol{w}^{(i,j)}\right)$, we have $\sum_{i=1}^n \boldsymbol{m}^{(i)}(X) = X\boldsymbol{v}$, where $X = \sum_{i=1}^n X^{(i)}$ and $\boldsymbol{v}^{(i)} = \sum_{i=1}^n \boldsymbol{v}^{(i)}$, therefore $\boldsymbol{m}^{(i)}(X)$ is the MAC share of P_i.

Since matrix-vector multiplication is a natural generalization of scalar-vector multiplication, a pair (P_i, P_j) can generate the additive sharing of $X^{(i)}\boldsymbol{v}^{(j)}$ by invoking m VOLE instances. In the k-th invocation of $\mathcal{F}^k_{\mathsf{VOLE}}$, P_i inputs the k-th column $\boldsymbol{x}_k^{(i)}$ of $X^{(i)}$ and P_j inputs the k-th component $v_k^{(j)}$ of global key share $\boldsymbol{v}^{(j)}$. According to the definition of VOLE, P_i receives $\boldsymbol{u}_k^{(i,j)}$ and P_j receives $\boldsymbol{w}_k^{(j,i)}$ such that $\boldsymbol{u}_k^{(j,i)} = v_k^{(j)}\boldsymbol{x}_k^{(i)} + \boldsymbol{w}_k^{(i,j)}$. By setting $\boldsymbol{u}^{(i,j)} = \sum_{k=1}^m \boldsymbol{u}_k^{(i,j)}$ and $\boldsymbol{w}^{(j,i)} = -\sum_{k=1}^m \boldsymbol{w}_k^{(j,i)}$, P_i and P_j jointly generate the additive sharing of $X^{(i)}\boldsymbol{v}^{(j)}$. It is easy to verify the correctness.

$$\begin{aligned}
\boldsymbol{u}^{(i,j)} + \boldsymbol{w}^{(j,i)} &= \sum_{k=1}^m \boldsymbol{u}_k^{(i,j)} - \boldsymbol{w}_k^{(j,i)} \\
&= \sum_{k=1}^m \boldsymbol{w}_k^{(i,j)} + v_k^{(j)}\boldsymbol{x}_k^{(i)} + \boldsymbol{w}_k^{(i,j)} \\
&= \sum_{k=1}^m v_k^{(j)}\boldsymbol{x}_k^{(i)}
\end{aligned}$$

Invoking VOLE alone is not sufficient to generate authenticated sharings in the presence of a malicious adversary. Because a corrupted party P_j may use inconsistent vectors $\left(\boldsymbol{x}_1^{(j)}, \cdots, \boldsymbol{x}_m^{(j)}\right)$ or vector $\boldsymbol{v}^{(j)}$ to interact with different honest parties. To prevent such attack, we introduce a consistency check which partially open a random linear combination of authenticated secret sharings to detect the corruption. To avoid leakage caused by this opening, we sacrifice a random authenticated sharing to mask the opened value. Although such a check can not guarantee the consistency of inputs in each invocation of $\mathcal{F}_{\mathsf{VOLE}}$, it guarantees that the sum of errors toward an honest party is zero, which suffices to generate the correct MAC share as errors cancel out after the addition.

Combining VOLE with consistency check, all parties can obtain the authenticated sharings. Protocol Π_{Auth} is the instantiation of functionality $\mathcal{F}_{Auth}$ which outputs the authenticated sharings.

Protocol 2: Π_{Auth}

- **Initialize**: If all parties receive (Init), each party P_i samples $\boldsymbol{v}^{(i)} \xleftarrow{\$} \mathcal{M}_{m\times 1}(\mathbb{F}_q)$ as global key share. For each ordered pair (P_i, P_j) and $k \in [m]$, P_i and P_j call the **Initialize** step of $\mathcal{F}^k_{\mathsf{VOLE}}$, where P_j inputs $v_k^{(j)}$.
- **Authenticate**: If all parties receive $(\mathsf{Auth}, [X_1], \ldots, [X_\ell])$:
  1. Each party P_i randomly samples a matrix $X_0^{(i)} \in \mathcal{M}_{m\times m}(\mathbb{F}_q)$.
  2. For $h \in \{0\} \cup [\ell]$, write $X_h^{(i)} = (\boldsymbol{x}_{h,1}^{(i)}, \cdots, \boldsymbol{x}_{h,m}^{(i)})$:
     (a) For each ordered pair (P_i, P_j) and $k \in [m]$, P_i and P_j call the **Multiply** step of $\mathcal{F}^k_{\mathsf{VOLE}}$, where P_i inputs $\boldsymbol{x}_{h,k}^{(i)}$.
     (b) P_i receives $\boldsymbol{u}_{h,k}^{(i,j)}$ and P_j receives $\boldsymbol{w}_{h,k}^{(j,i)}$ such that $\boldsymbol{u}_{h,k}^{(i,j)} = \boldsymbol{w}_{h,k}^{(j,i)} + v_k^{(j)} \boldsymbol{x}_{h,k}^{(i)}$.
     (c) Each party P_i sets $\boldsymbol{m}^{(i)}(X_h) = X_h^{(i)} \boldsymbol{v}^{(i)} + \sum_{j\neq i}\sum_{k\in[m]}(\boldsymbol{u}_{h,k}^{(i,j)} - \boldsymbol{w}_{h,k}^{(i,j)})$. Let $\left(X_h^{(i)}, \boldsymbol{v}^{(i)}, \boldsymbol{m}^{(i)}(X_h)\right)$ as the P_i's share of $\langle X_h \rangle$.
  3. Parties call $\mathcal{F}_{\mathsf{Coin}}$ ℓ times to obtain randomness $r_1, \cdots, r_\ell$.
  4. Parties locally compute $\langle Y \rangle = \langle X_0 \rangle + \sum_{h=1}^{\ell} r_h \langle X_h \rangle$.
  5. Parties invoke $Y' \leftarrow \pi_{\mathsf{Open}}(\langle Y \rangle)$ and $\pi_{\mathsf{check}}(Y', \langle Y \rangle)$ to check the correctness of opened value.
  6. If the check succeeds, output $\langle X_1 \rangle, \ldots, \langle X_\ell \rangle$.

Theorem 2. *Protocol Π_{Auth} securely implements $\mathcal{F}_{\mathsf{Auth}}$ in the ($\mathcal{F}_{\mathsf{VOLE}}$, $\mathcal{F}_{\mathsf{Coin}}$)-hybrid model.*

Proof. We analyze the consistency check in Π_{Auth} and defer the complete simulation-based security proof to the full version. There are two possible deviations in Π_{Auth}:

- A corrupted party P_j provides inconsistent global key share $\boldsymbol{v}^{(i)}$ with two different honest parties in the **Initialize** step.
- A corrupted party P_j provides inconsistent secret share $X_h^{(i)}$ for $h \in \{0\} \cup [\ell]$ with two different honest parties in the **Authentication** step.

In the command Auth, the adversary could introduce an arbitrarily additive error. For $h \in \{0\} \cup [\ell]$ and $k \in [m]$, let $\boldsymbol{x}_{h,k}^{(j,i)}, v_k^{(j,i)}$ be the *actual* input of P_j used in $\mathcal{F}_{\mathsf{VOLE}}^k$ with an honest party P_i. We fix an honest party P_{i_0}, and define the *correct* inputs $\boldsymbol{x}_{h,k}^{(j)}, v_k^{(j)}$ to be equal to $\boldsymbol{x}_{h,k}^{(j,i_0)}, v_k^{(j,i_0)}$ respectively. Then we obtain the additive error between actual inputs and correct inputs:

$$\boldsymbol{\delta}_{h,k}^{(j,i)} = \boldsymbol{x}_{h,k}^{(j,i)} - \boldsymbol{x}_{h,k}^{(j)} \qquad \epsilon_k^{(j,i)} = v_k^{(j,i)} - v_k^{(j)}$$

for each $j \in \mathcal{C}, i \notin \mathcal{C}$. For an honest party P_j, it keeps inputs $\boldsymbol{x}_{h,k}^{(j,i)} = \boldsymbol{x}_{h,k}^{(j)}$ and $v_k^{(j,i)} = v_k^{(j)}$ for each $i \neq j$. Finally, we define that for $i, j \in \mathcal{C}$, the additive error is zero, i.e., $\boldsymbol{\delta}_{h,k}^{(j,i)} = \mathbf{0}$ and $\epsilon_k^{(j,i)} = 0$.

For $j \in \mathcal{C}, i \notin \mathcal{C}$, if P_j behaves as sender and P_i behaves as receiver, we have that

$$\sum_{k=1}^{m} \left(\boldsymbol{u}_{h,k}^{(j,i)} - \boldsymbol{w}_{h,k}^{(i,j)} \right) = X_h^{(j)} \boldsymbol{v}^{(i)} + \Delta_h^{(j,i)} \boldsymbol{v}^{(i)}$$

where $\Delta_h^{(j,i)} = \left(\boldsymbol{\delta}_{h,1}^{(j,i)}, \cdots, \boldsymbol{\delta}_{h,m}^{(j,i)} \right)$. Similarly, reverse the role of P_i and P_j, we have that

$$\sum_{k=1}^{m} \left(\boldsymbol{u}_{h,k}^{(i,j)} - \boldsymbol{w}_{h,k}^{(j,i)} \right) = X_h^{(i)} \boldsymbol{v}^{(j)} + X_h^{(i)} \boldsymbol{\epsilon}^{(j,i)}$$

where $\boldsymbol{\epsilon}^{(j,i)} = \left(\epsilon_1^{(j,i)}, \cdots, \epsilon_m^{(j,i)} \right)^T$.

Sum up the MAC share $\boldsymbol{m}^{(i)}(X_h)$, we can see the following result:

$$\begin{aligned}
\sum_{i=1}^{n} \boldsymbol{m}^{(i)}(X_h) &= \sum_{i=1}^{n} X_h^{(i)} \boldsymbol{v}^{(i)} + \sum_{j \neq i} \sum_{k=1}^{m} \left(\boldsymbol{u}_{h,k}^{(i,j)} - \boldsymbol{w}_{h,k}^{(j,i)} \right) \\
&= \sum_{i=1}^{i} X_h^{(i)} \boldsymbol{v}^{(i)} + \sum_{j \neq i} X_h^{(i,j)} \boldsymbol{v}^{(j,i)} \\
&= X_h \boldsymbol{v} + \sum_{i \notin \mathcal{C}} \underbrace{\sum_{j \in \mathcal{C}} \Delta_h^{(j,i)}}_{\Delta_h^{(i)}} \boldsymbol{v}^{(i)} + \sum_{i \notin \mathcal{C}} X^{(i)} \underbrace{\sum_{j \in \mathcal{C}} \boldsymbol{\epsilon}^{(j,i)}}_{\boldsymbol{\epsilon}^{(i)}}
\end{aligned}$$

After the random linear combination with coefficients $(r_0 = 1, r_1, \cdots, r_\ell)$, we obtain the following MAC of variable Y:

$$\sum_{i=1}^{n} \boldsymbol{m}^{(i)}(Y) = Y \boldsymbol{v} + \sum_{i \notin \mathcal{C}} \sum_{h=0}^{\ell} r_h \Delta_h^{(i)} \boldsymbol{v}^{(i)} + \sum_{i \notin \mathcal{C}} \underbrace{\sum_{h=0}^{\ell} r_h X_h^{(i)}}_{Y^{(i)}} \boldsymbol{\epsilon}^{(i)}$$

Finally we proceed to check opening of Y. To pass the consistency, the adversary needs to introduce two errors $E = Y' - Y$ and $\boldsymbol{\gamma}$ such that:

$$\sum_{i=1}^{n} \boldsymbol{m}^{(i)}(Y) + \boldsymbol{\gamma} - (Y + E)\boldsymbol{v} = \boldsymbol{0}$$

$$\boldsymbol{\gamma} - E\boldsymbol{v} + \sum_{i \notin \mathcal{C}} \sum_{h=0}^{\ell} r_h \Delta_h^{(i)} \boldsymbol{v}^{(i)} + \sum_{i \notin C} Y^{(i)} \boldsymbol{\epsilon}^{(i)} = \boldsymbol{0}$$

$$\sum_{i \notin \mathcal{C}} \left(\sum_{h=0}^{\ell} r_h \Delta_h^{(i)} - E \right) \boldsymbol{v}^{(i)} + \sum_{i \notin C} Y^{(i)} \boldsymbol{\epsilon}^{(i)} = \sum_{i \in \mathcal{C}} E\boldsymbol{v}^{(i)} - \boldsymbol{\gamma}$$

We assert that if consistency check passes, then $\Delta_h^{(i)} = 0$ and $\boldsymbol{\epsilon}^{(i)} = \boldsymbol{0}$ with overwhelming probability. We prove this assertion with following two claims and defer their proofs in the full version [27].

Claim. If at least one $\boldsymbol{\epsilon}^{(i)} \neq \boldsymbol{0}$ for some $i \notin \mathcal{C}$, then consistency check passes with negligible probability.

Claim. If $\boldsymbol{\epsilon}^{(i)} = \boldsymbol{0}$ for all $i \notin \mathcal{C}$ and $\Delta_h^{(i)} \neq 0$ for some $i \notin \mathcal{C}$, then consistency check passes with negligible probability.

5 Preprocessing Phase

The preprocessing phase generates the authenticated random sharings $\langle R \rangle$ for the input gates, and the multiplication sextuples $(\langle A \rangle, \langle A^T \rangle, \langle B \rangle, \langle C \rangle, \langle R \rangle, \langle R^T \rangle)$ for the multiplication gates. In this section, we focus on multiplication sextuples. In Section A in the Supplementary Material, we describe the protocol Π_{Prep} for full-fledged preprocessing phase. To reduce the communication complexity of generating matrix triple, we introduce a variant of subfield VOLE called *vector oblivious product evaluation.* The process of generating multiplication sextuple is divided into two parts: the generation of Beaver triples $(\langle A \rangle, \langle B \rangle, \langle C \rangle)$ and double sharings $(\langle A \rangle, \langle A^T \rangle), (\langle R \rangle, \langle R^T \rangle)$.

5.1 Vector Oblivious Product Evaluation

A pseudorandom correlation generator (PCG) allows two parties to expand a pair of short, correlated seeds to a much larger amount of correlated randomness. Recently, efficient PCGs relying on several variants of learning parity with noise (LPN) assumptions were used to construct random VOLE (RVOLE) [10,11,17, 34–36]. While the communication complexity of original VOLE scales linearly in vector length, the communication complexity of PCG-based RVOLE is either the square root of vector length (under primal LPN assumption) [10,35,36] or logarithmic in vector length (under dual LPN assumption) [10,11,17,34]. In this work, we leverage the dual LPN assumption to reduce the communication cost.

In PCG-based RVOLE, the sender P_A sends a seed $s \in S$ instead of a whole vector $\boldsymbol{x}$, where S is the space of seeds. The property *programmability* was introduced to PCG-based RVOLE in [12], which allows the sender to reuse its seed s in different instances of RVOLE. We model the programmability with function $\mathsf{Expand} : S \to \mathbb{F}_q^a$, which deterministically expands the given random seed to a pseudorandom vector of given length a over $\mathbb{F}_q$.

Boyle et al. [12], proposed a variant of RVOLE, called subfield VOLE, which can securely compute $\boldsymbol{u} = v\boldsymbol{x} + \boldsymbol{w}$, where $\boldsymbol{x} \in \mathbb{F}_q^a, v \in \mathbb{F}_{q^b}, \boldsymbol{u}, \boldsymbol{w} \in \mathbb{F}_{q^b}^a$. In a subfield VOLE instance between P_A and P_B, $\boldsymbol{x} \in \mathbb{F}_q^a$ is generated from a seed $s \in S$ chosen by P_A and $v \in \mathbb{F}_{q^b}$ is directly chosen by P_B. Thus, subfield VOLE could be regarded as a PCG for product of vectors, i.e., rewrite $v \in \mathbb{F}_{q^b}$ as $\boldsymbol{v} \in \mathbb{F}_q^b$ and the subfield VOLE actually computes the additive sharing of $\boldsymbol{x} \otimes \boldsymbol{v} \in \mathcal{M}_{a \times b}(\mathbb{F}_q)$. Since we have already shown that the product of two $m \times m$ matrices can be decomposed into the sums of products of the form $\boldsymbol{x} \otimes \boldsymbol{v} \in \mathcal{M}_{m \times m}(\mathbb{F}_q)$, it suffices to invoke subfield VOLE m times to compute the matrix multiplication.

However, in subfield VOLE, the input $v \in \mathbb{F}_{q^b}$ is chosen uniformly at random which means the input size of P_B is $b \log q$ bits. Note that in our setting, a and b are of almost the same size $\Omega(m)$ which means it is necessary to minimize the input size from both sides. Thus, we modify this subfield VOLE by generating a pseudorandom element $v \in \mathbb{F}_{q^b}$ from a seed. We call this modified subfield VOLE vector oblivious product evaluation (VOPE). The functionality $\mathcal{F}_{\mathsf{VOPE}}^{a,b}$ can be found in Functionality 6. The instantiation of $\mathcal{F}_{\mathsf{VOPE}}^{a,b}$ is given in the full version [27], which is adapted from [33].

Functionality 6: $\mathcal{F}_{\mathsf{VOPE}}^{a,b}$

Let $\mathsf{Expand} : S \to \mathbb{F}_q^a, \mathsf{Expand}' : S' \to \mathbb{F}_q^b$ be the deterministic expansion functions with seed space S, S' and output length a, b, respectively. The functionality runs between sender P_A and receiver P_B.

Upon receiving $s \in S$ from P_A and $s' \in S'$ from P_B:

1. Compute $\boldsymbol{x} = \mathsf{Expand}(s), \boldsymbol{v} = \mathsf{Expand}'(s')$.
2. Sample $W \overset{\$}{\leftarrow} \mathcal{M}_{a,b}(\mathbb{F}_q)$. If P_B is corrupted, receive W from the adversary.
3. Compute $U = \boldsymbol{x} \otimes \boldsymbol{v} - W$. If P_A is corrupted, receive U from adversary and recompute $W = \boldsymbol{x} \otimes \boldsymbol{v} - U$.
4. Output U to P_A and W to P_B.

5.2 Generation of Beaver Triple

To simplify our proof, recall that we define $\boldsymbol{u} \otimes \boldsymbol{v} = \boldsymbol{u}\boldsymbol{v}^T$. The first step of generating Beaver triple is to securely compute matrix multiplication, which can be decomposed into some tensor products of vectors. Assume that there are two random matrices $A \in \mathcal{M}_{m_1 \times m_2}(\mathbb{F}_q), B \in \mathcal{M}_{m_2 \times m_3}(\mathbb{F}_q)$. Let $\boldsymbol{a}_i$ be the i-th column of A and $\boldsymbol{b}_i$ be the i-th row of B. Then, we have $AB = \sum_{i=1}^{m_2} \boldsymbol{a}_i \otimes \boldsymbol{b}_i$. This implies that it suffices to compute m_2 products $C_i = \boldsymbol{a}_i \otimes \boldsymbol{b}_i \in \mathcal{M}_{m_1 \times m_3}(\mathbb{F}_q)$, and then add them together to obtain $AB = C = \sum_{i \in [m_2]} C_i$.

Procedure π_{Mult} outputs the authenticated Beaver triples. Note that seeds s, s' will be reused several times for different pairs of parties. A corrupted party could cause the inconsistency of seeds towards different honest parties. Therefore, we add a consistency check at the end of π_{Mult}: To check the correctness of $(\langle A \rangle, \langle B \rangle, \langle C \rangle)$, we sacrifice another Beaver triple $(\langle A' \rangle, \langle B \rangle, \langle C' \rangle)$.

Procedure 3: π_{Mult}

Let $\mathsf{Expand} : S \to \mathbb{F}_q^{2m}, \mathsf{Expand}' : S' \to \mathbb{F}_q^m$ be the deterministic expansion functions with seed space S, S' and output length a, b, respectively. The procedure generates an authenticated triple $(\langle A \rangle, \langle B \rangle, \langle C \rangle)$ where $A, B \xleftarrow{\$} \mathcal{M}_{m \times m}(\mathbb{F}_q)$ and $C = AB$.

- **Multiply:**
  1. Each party P_i samples seeds $\left(s_1^{(i)}, \cdots, s_m^{(i)}\right) \in S^m, \left({s'_1}^{(i)}, \cdots, {s'_m}^{(i)}\right) \in {S'}^m$ and obtains $\hat{A}^{(i)} = \left(\hat{\boldsymbol{a}}_1^{(i)}, \cdots, \hat{\boldsymbol{a}}_m^{(i)}\right) \in \mathcal{M}_{2m \times m}(\mathbb{F}_q), B^{(i)} = \left(\boldsymbol{b}_1^{(i)}, \cdots, \boldsymbol{b}_m^{(i)}\right)^T$, where $\hat{\boldsymbol{a}}_k^{(i)} = \mathsf{Expand}\left(s_k^{(i)}\right), \boldsymbol{b}_k^{(i)} = \mathsf{Expand}'({s'_k}^{(i)})$ for $k \in [m]$.
  2. For $k \in [m]$ and each ordered pair (P_i, P_j):
     (a) P_i and P_j invoke $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$, where P_i inputs $s_k^{(i)}$ and P_j inputs ${s'_k}^{(j)}$.
     (b) P_i receives $U_k^{(i,j)}$ and P_j receives $W_k^{(j,i)}$
  3. Each party P_i locally computes
$$\hat{C}^{(i)} = \hat{A}^{(i)} B^{(i)} + \sum_{j \neq i} \sum_{k \in [m]} \left(U_k^{(i,j)} + W_k^{(i,j)}\right)$$
  4. Each party P_i rewrites: $\hat{A}^{(i)} = \begin{pmatrix} A^{(i)} \\ {A'}^{(i)} \end{pmatrix}, \hat{C}^{(i)} = \begin{pmatrix} C^{(i)} \\ {C'}^{(i)} \end{pmatrix}$ and obtain $[A], [A'], [B], [C], [C']$.
- **Authenticate**: All parties invoke $\mathcal{F}_{\mathsf{Auth}}$ to obtain $\langle A \rangle, \langle A' \rangle, \langle B \rangle, \langle C \rangle$ and $\langle C' \rangle$.
- **Sacrifice**:
  1. All parties invoke $\mathcal{F}_{\mathsf{Coin}}$ to obtain a random element χ.
  2. All parties locally compute $\langle D \rangle = \chi \langle A \rangle - \langle A' \rangle$ and partially open $D \leftarrow \pi_{\mathsf{Open}}(\langle D \rangle)$.
  3. All parties locally compute $\langle E \rangle = \chi \langle C \rangle - \langle C' \rangle - D \langle B \rangle$ and partially open $E \leftarrow \pi_{\mathsf{Open}}(\langle E \rangle)$.
  4. If $E \neq 0$, then aborts.
- **Output**: If no party aborts, all parties output $(\langle A \rangle, \langle B \rangle, \langle C \rangle)$.

5.3 Generation of Double Sharing

To generate ℓ multiplication sextuples for securely computing ℓ multiplication gates, we need $2\ell+1$ double sharings of the form $\langle A\rangle, \langle A^T\rangle$ with some random matrix A. Procedure π_{Double} receives the authenticated sharings $\langle A\rangle$ and output the pair of authenticated sharing $(\langle A\rangle, \langle A^T\rangle)$. We briefly explain the idea of this procedure. Observe that $[A^T]$ can be obtained by locally applying the transpose to each share of $[A]$. Then, we apply the $\mathcal{F}_{\mathsf{Auth}}$ to obtain the authenticated sharing $\langle A^T\rangle$. Take random linear combinations of $2\ell+1$ double sharings $(\langle A_i\rangle, \langle A_i^T\rangle)$ respectively and partially open them to C and D. If there is no corruption, $C = D^T$ and the check passes. Otherwise, this check will pass with probability at most $1/q$.

Procedure 4: π_{Double}

Let n_D denote th number of double sharings. The procedure produces n_D pairs of authenticated sharing $\langle A_i\rangle, \langle A_i^T\rangle, i \in [n_D]$.
Double: Upon receiving (Double, $\langle A_1\rangle, \ldots, \langle A_{n_D}\rangle$) from all parties:

1. All parties invoke π_{Rand} to obtain $[A_0]$.
2. All parties locally compute $[A_i^T]$ from $[A_i]$ for $i \in \{0\} \cup [n_D]$ by taking the transpose of each share.
3. All parties invoke $\mathcal{F}_{\mathsf{Auth}}$ with command (Auth, $[A_0], [A_0^T], [A_1^T], \ldots, [A_{n_D}^T]$) to obtain the authenticated sharings $\langle A_0\rangle, \langle A_0^T\rangle, \langle A_1^T\rangle, \ldots, \langle A_{n_D}^T\rangle$.
4. All parties call $\mathcal{F}_{\mathsf{Coin}}$ n_D times to obtain $r_1, \cdots, r_{n_D}$.
5. All parties locally compute
$$\langle C\rangle = \sum_{i=1}^{n_D} r_i\langle A_i\rangle + \langle A_0\rangle \quad \langle D\rangle = \sum_{i=1}^{n_D} r_i\langle A_i^T\rangle + \langle A_0^T\rangle$$
6. All parties invoke π_{Open} to partially open C and D.
7. If $C \neq D^T$, then aborts.
8. All parties invoke π_{Check} to check the opened values.
9. If no party aborts, output n_D pairs of authenticated sharings $(\langle A_i\rangle, \langle A_i^T\rangle), i \in [n_D]$.

Putting Together. Protocol Π_{Sextuple} instantiates the functionality $\mathcal{F}_{\mathsf{Sextuple}}$ by invoking the procedures introduced above. π_{Mult} and π_{Double} are used to produce the authenticated sharings $(\langle A\rangle, \langle B\rangle, \langle C\rangle)$ and $(\langle A\rangle, \langle A^T\rangle), (\langle R\rangle, \langle R^T\rangle)$, respectively.

Protocol 5: Π_{Sextuple}

This protocol produces ℓ authenticated sextuples $(\langle A\rangle, \langle A^T\rangle, \langle B\rangle, \langle C\rangle, \langle R\rangle, \langle R^T\rangle)$ with $C = AB$:

1. All parties invoke π_{Mult} ℓ times to produce $(\langle A_i\rangle, \langle B_i\rangle, \langle C_i\rangle)$ with $C_i = A_i B_i$ for $i \in [\ell]$.
2. All parties invoke π_{Rand} ℓ times to obtain $[R_1], \ldots, [R_\ell]$.
3. All parties invoke $\mathcal{F}_{\mathsf{Auth}}$ with command $(\mathsf{Auth}, [R_1], \ldots, [R_\ell])$ to obtain $\langle R_i\rangle$ for $i \in [\ell]$.
4. All parties set $n_D = 2\ell$ and invoke π_{Double} with command $(\mathsf{Double}, \langle A_1\rangle, \ldots, \langle A_\ell\rangle, \langle R_1\rangle, \ldots, \langle R_\ell\rangle)$ to obtain $(\langle A_i\rangle, \langle A_i^T\rangle)$ and $(\langle R_i\rangle, \langle R_i^T\rangle)$ for $i \in [\ell]$.
5. Output $(\langle A_i\rangle, \langle A_i^T\rangle, \langle B_i\rangle, \langle C_i\rangle, \langle R_i\rangle, \langle R_i^T\rangle)$ for $i \in [\ell]$.

Theorem 3. *Protocol Π_{Sextuple} securely implements $\mathcal{F}_{\mathsf{Sextuple}}$ in the ($\mathcal{F}_{\mathsf{Auth}}$, $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$, $\mathcal{F}_{\mathsf{Coin}}$)-hybrid model.*

Proof. Let $\mathcal{Z}$ be the environment, which we also refer to as the adversary capable of corrupting a set $\mathcal{C}$ containing at most $n-1$ parties. We construct a simulator $\mathcal{S}$ such that the real execution and ideal execution are indistinguishable to $\mathcal{Z}$. Here we only prove the security of π_{Mult} and refer to the full version [27] for the complete proof.

In functionality $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$ between P_i and P_j, both P_i and P_j only input their seeds. Therefore, the corrupted parties can only choose inconsistent seeds for different honest parties, which can not translate to an arbitrarily chosen additive error. However, for the convenience of analysis, we follow the idea of [33] and improve the ability of adversary to introduce an arbitrarily chosen additive error.

Simulating the Multiply Step. The simulator $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$. For $j \in \mathcal{C}$ and $i \notin \mathcal{C}$, let $s_k^{(j,i)}$ and ${s'_k}^{(j,i)}$ be the *actual* input in the k-th invocation of $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$ for $k \in [m]$. Fix an honest party P_{i_0} and define the *correct* input $s_k^{(j)}$ and ${s'_k}^{(j)}$ to be equal to $s_k^{(j,i_0)}$ and ${s'_k}^{(j,i_0)}$, respectively. For $i \notin \mathcal{C}$, $\mathcal{S}$ randomly samples $\hat{A}^{(i)} \xleftarrow{\$} \mathcal{M}_{\tau m\times m}(\mathbb{F}_q), B^{(i)} \xleftarrow{\$} \mathcal{M}_{m\times m}(\mathbb{F}_q)$. For $j \in \mathcal{C}$, $\mathcal{S}$ receives ${s_k}^{(j,i)}$ and ${s'_k}^{(j,i)}$ from the adversary, where $i \notin \mathcal{C}, k \in [m]$. Then $\mathcal{S}$ receives $\left\{U_k^{(j,i)}, W_k^{(j,i)}\right\}_{j\in\mathcal{C}, i\notin\mathcal{C}}$ from the adversary and recomputes $\left\{U_k^{(i,j)}, W_k^{(i,j)}\right\}_{i\notin\mathcal{C}, j\in\mathcal{C}}$ accordingly. Finally, $\mathcal{S}$ honestly computes $\hat{C}^{(i)}$.

Simulating the Authentication Step. $\mathcal{S}$ emulates functionality $\mathcal{F}_{\mathsf{Auth}}$ with inputs from corrupted parties controlled by $\mathcal{Z}$. $\mathcal{S}$ authenticates additive sharings and we denote by E_{Auth}, E'_{Auth} errors introduced in the authentication step. If E_{Auth}, E'_{Auth} are not zero, then the authenticated values are different from those in the previous step. If $\mathcal{Z}$ sends Abort to $\mathcal{F}_{\mathsf{Auth}}$, $\mathcal{S}$ sends Abort to $\mathcal{F}_{\mathsf{Sextuple}}$.

Simulating the Sacrifice Step. $\mathcal{S}$ samples $D \leftarrow \mathcal{M}_{m\times m}(\mathbb{F}_q)$ as $\chi A - A'$. If the triple is incorrect, $\mathcal{S}$ aborts; otherwise, $\mathcal{S}$ outputs it as a valid triple.

Indistinguishability. We argue that $\mathcal{Z}$ cannot distinguish real execution and simulated one. We will show that if no abort happens, the probability that adversary introduces some non-zero errors is negligible and the distribution of opened value is statistically close in both of the worlds.

Now we proceed to the introduced errors during **Multiply** step. Let $\hat{A}^{(j,i)}$ and $B^{(j,i)}$ be the matrices generated by seeds $\left(s_1^{(j,i)}, \cdots, s_m^{(j,i)}\right)$ and $\left({s'_1}^{(j,i)}, \cdots, {s'_m}^{(j,i)}\right)$, respectively. In the k-th invocation of $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$, denote the errors as $\hat{\boldsymbol{\delta}}_k^{(j,i)} = \hat{\boldsymbol{a}}_k^{(j,i)} - \hat{\boldsymbol{a}}_k^{(j)}$ and $\boldsymbol{\gamma}_k^{(j,i)} = \boldsymbol{b}_k^{(j,i)} - \boldsymbol{b}_k^{(j)}$. Then we conclude that for $k \in [m], i \notin \mathcal{C}$ and $j \in \mathcal{C}$:

$$U_k^{(i,j)} + W_k^{(j,i)} = \hat{\boldsymbol{a}}_k^{(i)} \otimes \boldsymbol{b}^{(j)} + \hat{\boldsymbol{a}}_k^{(i)} \otimes \boldsymbol{\gamma}_k^{(j,i)}$$
$$U_k^{(j,i)} + W_k^{(i,j)} = \hat{\boldsymbol{a}}_k^{(j)} \otimes \boldsymbol{b}^{(i)} + \hat{\boldsymbol{\delta}}_k^{(j,i)} \otimes \boldsymbol{b}_k^{(i)}$$

Following similar analysis in the proof of Theorem 2, we define $\hat{\Delta}^{(j,i)} = \left(\hat{\boldsymbol{\delta}}_1^{(j,i)}, \cdots, \hat{\boldsymbol{\delta}}_m^{(j,i)}\right)$, $\Gamma^{(j,i)} = \left(\boldsymbol{\gamma}_1^{(j,i)}, \cdots, \boldsymbol{\gamma}_m^{(j,i)}\right)^T$ and compute $\hat{C}$ as

$$\begin{aligned}
\hat{C} = \sum_{i\in[n]} \hat{C}^{(i)} &= \hat{A}B + \sum_{i\notin\mathcal{C}} \sum_{j\in\mathcal{C}} \sum_{k\in[m]} \left(\hat{\boldsymbol{a}}_k^{(i)} \otimes \boldsymbol{\epsilon}_k^{(j,i)} + \hat{\boldsymbol{\delta}}_k^{(j,i)} \otimes \boldsymbol{b}_k^{(i)}\right) \\
&= \hat{A}B + \sum_{i\notin\mathcal{C}} \sum_{j\in\mathcal{C}} \hat{A}^{(i)} \Gamma^{(j,i)} + \hat{\Delta}^{(j,i)} B^{(i)} \\
&= \hat{A}B + \sum_{i\notin\mathcal{C}} \hat{A}^{(i)} \Gamma^{(i)} + \hat{\Delta}^{(i)} B^{(i)}
\end{aligned}$$

where $\hat{\Delta}^{(i)} = \sum_{j\in\mathcal{C}} \hat{\Delta}^{(j,i)}$ and $\Gamma^{(i)} = \sum_{j\in\mathcal{C}} \Gamma^{(j,i)}$. Splitting the matrices into 2 blocks, we have that:

$$\begin{pmatrix} C \\ C' \end{pmatrix} = \begin{pmatrix} A \\ A' \end{pmatrix} B + \sum_{i\notin\mathcal{C}} \begin{pmatrix} A^{(i)} \\ {A'}^{(i)} \end{pmatrix} \Gamma^{(i)} + \begin{pmatrix} \Delta^{(i)} \\ {\Delta'}^{(i)} \end{pmatrix} B^{(i)}$$

After the **Authentication** step, all parties obtain $\langle A\rangle, \langle A'\rangle, \langle B\rangle, \langle C\rangle, \langle C'\rangle$. Assume that the adversary introduces additive error E_{Auth}, E'_{Auth} in this step, then A, A', B, C, C' satisfy that:

$$C = AB + E_1 + E_2 + E_{Auth}$$
$$C' = A'B + E'_1 + E'_2 + E'_{Auth}$$

and

$$E_1 = \sum_{i\notin\mathcal{C}} A^{(i)} \Gamma^{(i)} \qquad E_2 = \sum_{i\notin\mathcal{C}} \Delta^{(i)} B^{(i)}$$
$$E'_1 = \sum_{i\notin\mathcal{C}} {A'}^{(i)} \Gamma^{(i)} \qquad E_2 = \sum_{i\notin\mathcal{C}} {\Delta'}^{(i)} B^{(i)}$$

If no abort happens in the **Sacrifice** step, we come to the following conclusions and defer their proofs to the full version [27].

Claim. If the sacrifice step passes, then $E = E_1 + E_2 + E_{Auth} = 0$ and $E' = E'_1 + E'_2 + E'_{Auth}$ with overwhelming probability.

Claim. If the sacrifice step passes, then $\{\Gamma^{(i)}, \Delta^{(i)}, {\Delta'}^{(i)}\}_{i \notin \mathcal{C}}$ are zero with overwhelming probability.

Finally, we want to show that the opened value D in the real execution is computationally indistinguishable from the uniform matrix in the simulated execution. Given that $D = \chi A - A'$, it suffices to prove A' looks uniformly random to $\mathcal{Z}$ and thus can serve as a mask. Each column $\boldsymbol{a}'_i$ of A' is part of output of expansion function Expand, therefore we want to show that Expand acts as a PRG. The concrete construction of Expand is given in the full version [27], and the pseudorandomness of output is guaranteed by dual LPN assumption.

6 Analysis

In this section, we analyze the communication and computation cost of our MPC protocol over $\mathcal{M}_{m \times m}(\mathbb{F}_q)$ assuming $q \geq 2^\kappa$. The computation complexity is measured by the number of multiplications.

6.1 Analysis of the Online Phase

First, we consider communication complexity. At each step of partial opening a matrix, all parties send their shares to a specific party, then let this party reconstruct and broadcast the secret, thus the communication complexity is $2m^2(n-1)\log q$ bits. For each multiplication gate, all parties need to partially open three shares $\langle D \rangle, \langle E \rangle, \langle F \rangle$ and thus the communication complexity is $6m^2(n-1)\log q$ bits. Each input gate requires P_i to broadcast the difference between X and mask R, which communicates $m^2(n-1)\log q$ bits. For the output gate, the partial opening needs $2m^2(n-1)\log q$ bit of communication and verification needs $mn^2 \log q$ bits of communication via simultaneous message channel.

Now we proceed to analyze the computation complexity for each multiplication gate. According to Mult command in Π_{Online}, all parties execute the following computation: $E^T[A^T], E^T[\![A^T\boldsymbol{v}]\!], D[B], D[\![B\boldsymbol{v}]\!], DE, DE[\![\boldsymbol{v}]\!]$. Since left multiplication requires m^3 and m^2 multiplications in scheme $[\cdot]$ and $[\![\cdot]\!]$ respectively, the overall computation complexity is $3m^3 + 3m^2$ multiplications.

Another measure is share size, which is $m(m+1)n\log q$ bits, since $[\![\boldsymbol{v}]\!]$ remains unchanged in each authenticated sharing and we omit this item.

We analyze the communication complexity, computation complexity and share size of other MPC protocols and list the results in Table 1. Here FI and FD refer to the online communication with function-independent and function-dependent preprocessing, respectively. Although our protocol needs slightly more communication than [16], our protocol has the smallest share size and computation complexity among these protocols. Moreover, the improvement of our MPC protocol by resorting to function-dependent preprocessing can achieve the same communication complexity as [16].

6.2 Analysis of the Preprocessing Phase

The task of preprocessing is to generate random sharings and multiplication sextuples. The communication cost is mainly caused by Π_{Sextuple} which produces the multiplication sextuples. As our preprocessing phase uses VOLE and VOPE as the building blocks, we calculate the communication cost of preprocessing phase in terms of the calls of the functionality $\mathcal{F}_{\mathsf{VOLE}}$ and $\mathcal{F}_{\mathsf{VOPE}}$.

To generate a random authenticated sharing $\langle R \rangle$ for an input gate, where the secret R is known to P_i, P_i distributes the additive share $R^{(j)}$ to P_j and invokes $\mathcal{F}_{\mathsf{VOLE}}$ with P_j. After producing $\ell + 1$ such random sharings, all parties invoke π_{Check} to check the consistency of these sharings. If ℓ is large enough, the communication cost of the consistency check can be amortized away. In this case, the preparation for an input gate requires $n-1$ calls of $\mathcal{F}_{\mathsf{VOLE}}$.

Protocol Π_{Sextuple} produces ℓ sextuples by generating ℓ Beaver triples and 2ℓ double sharings. During this process, the communication cost is caused by ℓ calls of Π_{Auth} and the invocation of procedure π_{Mult} and π_{Double}. Procedure π_{Mult} generates a multiplication triple $(\langle A \rangle, \langle B \rangle, \langle C \rangle)$ by making $mn(n-1)$ calls of $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$, 5 calls of Π_{Auth} and 2 calls of π_{open}. Procedure π_{Double} generates 2ℓ authenticated sharings $\langle A \rangle, \langle A^T \rangle$ by making $2\ell + 2$ calls of Π_{Auth} and 2 calls of π_{open}. In summary, generating a sextuple requires $mn(n-1)$ calls of $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$ and $8mn(n-1)$ calls of $\mathcal{F}_{\mathsf{VOLE}}$ assuming ℓ is large enough.

The communication cost of $\mathcal{F}_{\mathsf{VOLE}}$ scales linearly in the length of the vector, which incurs $O(m \log q)$ bits of communication. The analysis of $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$ depends on the dual LPN parameters. Given the dual LPN parameter $(2m, 2cm, t)$, $\mathcal{F}_{\mathsf{VOPE}}^{2m,m}$ requires t invocations of $\mathcal{F}_{\mathsf{rsVOLE}}^{m}$ (which is sublinear in m), $t \log \frac{2cm}{t}$ invocations of κ-bit OT and exchange of $t(1+m)$ field elements, which result in $O(m \log q)$ bits of communication. (Note that $t = O(1)$ which does not grow with m.)

Now we proceed to the analysis of the concrete communication cost. We pick the parameters in [16] for a comparison. For a matrix ring $\mathcal{M}_{128 \times 128}(\mathbb{F}_q)$ where the prime number q satisfies $\log q \approx 128$, [16] shows that each party communicates 12.46MB to generate a matrix triple for the multiplication gate. Our protocol requires 2^7 invocations of $\mathcal{F}_{\mathsf{VOPE}}^{2^8,2^7}$ and 2^{10} invocations of $\mathcal{F}_{\mathsf{VOLE}}^{2^7}$. We choose the security parameter to be 80 bits and then obtain the corresponding

Table 1. The comparison of MPC protocols over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ in terms of share size, communication complexity and computation complexity of a multiplication gate.

	communication	share size	# multiplications
SPDZ [20]	$4m^3(n-1)\log q$	$2m^2 \log q$	$6m^3$
matrix triple [16]	$4m^2(n-1)\log q$	$2m^2 \log q$	$5m^3 + m^2$
This work (FI)	$6m^2(n-1)\log q$	$m(m+1)\log q$	$3m^3 + 3m^2$
This work (FD)	$4m^2(n-1)\log q$	$m(m+1)\log q$	$3m^3 + 3m^2$

dual LPN parameters in [26]. The detailed calculation of communication cost of $\mathcal{F}_{\mathsf{VOPE}}$ and $\mathcal{F}_{\mathsf{VOLE}}$ is deferred to the full version [27].

Table 2 demonstrates the communication cost of our protocol, the protocol relying on the random VOLE [10], the protocol relying on subfield VOLE [33] and the protocol relying on the homomorphic encryption [16] to prepare the correlated randomness for computing the multiplication gate. The "random VOLE" protocol computes random matrix multiplication with m^2 random VOLE instances [10], and the "subfield VOLE" protocol invokes m times of subfield VOLE in [33], where the extension field is defined as $\mathbb{F}_{q^m}$. One can see that the communication cost of our protocol grows more slowly than [16]. The reason is that the amortized communication cost of PCG-based VOLE decreases with the size of m.

Table 2. The communication cost to prepare correlated randomness for computing a multiplication gate.

m	random VOLE	subfield VOLE	This work	HE [16]
128	83.5 MB	34.8 MB	19.0 MB	12.5 MB
256	362 MB	138 MB	60 MB	50 MB
512	1453 MB	518 MB	198 MB	199 MB
1024	6000 MB	2004 MB	739 MB	797 MB

6.3 Experimental Result

Online Phase. We implement the online phase of different MPC protocols over $\mathcal{M}_{m\times m}(\mathbb{F}_q)$ in C++ with the multiple precision integer arithmetic provided by MPIR library [6]. All experiments were carried out on a server equipped with an Intel Xeon Gold 5220R processor and 128GB RAM. We apply Linux tc command to emulate a real network environment and simulate the LAN network with 1Gbps bandwidth, 1ms latency. Table 3 compares the performances of computing a multiplication gate for each MPC protocol, which shows that our approach is around 1.38x-1.85x faster than [16].

Table 3. Runtime to compute a multiplication gate in the online phase.

m	SPDZ [20]	matrix triple [16]	This work
128	1.3 sec	96 ms	52 ms
256	9.5 sec	559 ms	329 ms
512	77.9 sec	5.0 sec	3.2 sec
1024	633 sec	42.5 sec	30.9 sec

Preprocessing Phase. We present the benchmarks of VOLE-based preprocessing protocols to generate the correlated randomness for a multiplication gate, in which secure random matrix multiplication is the bottleneck of the computation. All VOLE-based preprocessing protocols rely on PCG techniques, which expand a pair of short seeds to long correlated randomness. We apply the PCG implementation of libOTe [32] to estimate the runtime of the expansion step, which is based on quasi-cyclic codes in [12]. To estimate the efficiency of generating seeds, we calculate the required number of VOLEs and OTs and benchmark the runtime of VOLE and OT with Lattigo [1] and libOTe [32] libraries, respectively. The cost of VOLE is estimated with the ring-LWE based OLE protocol in [7].

Table 4 provides total estimated runtime on secure random matrix multiplication in the LAN setting. To make a fair comparison with [16], all VOLE-based protocols are tested with 16 threads. As can be seen from the table, our preprocessing phase achieves a 1.44x-24.17x speedup compared to [16] with the same thread number. It is noteworthy that [16] requires a key generation and a one-time setup (when $m = 128$, these operations take around 83 seconds and 14.5 seconds respectively), while our protocol does not rely on a heavy setup. We provide a full-fledged experiment result of preprocessing in the full version [27].

Table 4. Benchmark: Runtime to prepare correlated randomness for computing a multiplication gate, measured with 16 threads.

m	random VOLE	subfield VOLE	This work	HE [16]
128	3.6 sec	2.9 sec	4.1 sec	5.9 sec
256	13.9 sec	10.1 sec	8.2 sec	25.5 sec
512	56.4 sec	38.2 sec	16.8 sec	2.3 min
1024	4.1 min	2.5 min	36.0 sec	14.5 min

Acknowledgement. The authors would like to thank Jiawei Ni for her assistance with implementation. We are also grateful for valuable suggestions from anonymous reviewers in Asiacrypt 2024. The work was supported in part by the National Key Research and Development (R&D) Program of China under Grant 2022YFA1004900 and in part by the National Natural Science Foundation of China under Grants 12031011, 12361141818, and 12101404. This work was supported in part by Natural Science Foundation of Shanghai under the 2024 Shanghai Action Plan for Science, Technology and Innovation Grant 24BC3200700. This work was also supported in part by Ant Group through CCF-Ant Research Fund CCF-AFSG RF20230306.

References

1. Lattigo v5. Online: https://github.com/tuneinsight/lattigo (Nov 2023), ePFL-LDS, Tune Insight SA

2. Abspoel, M., Cramer, R., Damgård, I., Escudero, D., Rambaud, M., Xing, C., Yuan, C.: Asymptotically good multiplicative LSSS over galois rings and applications to MPC over $\mathbb{Z}/p^k\mathbb{Z}$. In: ASIACRYPT 2020. LNCS, vol. 12493, pp. 151–180. Springer (2020)
3. Abspoel, M., Cramer, R., Damgård, I., Escudero, D., Yuan, C.: Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In: TCC 2019. LNCS, vol. 11891, pp. 471–501. Springer (2019)
4. Applebaum, B., Damgård, I., Ishai, Y., Nielsen, M., Zichron, L.: Secure arithmetic computation with constant computational overhead. In: CRYPTO 2017. LNCS, vol. 10401, pp. 223–254. Springer (2017)
5. Applebaum, B., Konstantini, N.: Actively secure arithmetic computation and VOLE with constant computational overhead. In: EUROCRYPT 2023. LNCS, vol. 14005, pp. 190–219. Springer (2023)
6. B. Gladman, W.H., J. Moxham, e.a.: MPIR: Multiple Precision Integers and Rationals (2015), version 2.7.0, http://mpir.org
7. Baum, C., Escudero, D., Pedrouzo-Ulloa, A., Scholl, P., Troncoso-Pastoriza, J.R.: Efficient protocols for oblivious linear function evaluation from ring-lwe. J. Comput. Secur. **30**(1), 39–78 (2022)
8. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: CRYPTO '91. LNCS, vol. 576, pp. 420–432. Springer (1991)
9. Ben-Efraim, A., Nielsen, M., Omri, E.: Turbospeedz: Double your online spdz! improving SPDZ using function dependent preprocessing. In: ACNS 2019. LNCS, vol. 11464, pp. 530–549. Springer (2019)
10. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y.: Compressing vector OLE. In: ACM CCS 2018. pp. 896–912. ACM (2018)
11. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Resch, N., Scholl, P.: Correlated pseudorandomness from expand-accumulate codes. In: CRYPTO 2022. LNCS, vol. 13508, pp. 603–633. Springer (2022)
12. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: CRYPTO 2019. LNCS, vol. 11694, pp. 489–518. Springer (2019)
13. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators from ring-lpn. In: CRYPTO 2020. LNCS, vol. 12171, pp. 387–416. Springer (2020)
14. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer (2012)
15. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS 2001. pp. 136–145. IEEE Computer Society (2001)
16. Chen, H., Kim, M., Razenshteyn, I.P., Rotaru, D., Song, Y., Wagh, S.: Maliciously secure matrix multiplication with applications to private deep learning. In: ASIACRYPT 2020. LNCS, vol. 12493, pp. 31–59. Springer (2020)
17. Couteau, G., Rindal, P., Raghuraman, S.: Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In: CRYPTO 2021. LNCS, vol. 12827, pp. 502–534. Springer (2021)
18. Cramer, R., Damgård, I., Escudero, D., Scholl, P., Xing, C.: Spd$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In: CRYPTO 2018. LNCS, vol. 10992, pp. 769–798. Springer (2018)
19. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In: ESORICS 2013. LNCS, vol. 8134, pp. 1–18. Springer (2013)

20. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer (2012)
21. Escudero, D., Goyal, V., Polychroniadou, A., Song, Y., Weng, C.: Superpack: Dishonest majority MPC with constant online communication. In: EUROCRYPT 2023. LNCS, vol. 14005, pp. 220–250. Springer (2023)
22. Escudero, D., Soria-Vazquez, E.: Efficient information-theoretic multi-party computation over non-commutative rings. In: CRYPTO 2021. LNCS, vol. 12826, pp. 335–364. Springer (2021)
23. Escudero, D., Xing, C., Yuan, C.: More efficient dishonest majority secure computation over $\mathbb{Z}_{2^k}$ via galois rings. In: CRYPTO 2022. LNCS, vol. 13507, pp. 383–412. Springer (2022)
24. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. p. 144 (2012), http://eprint.iacr.org/2012/144
25. Jiang, X., Kim, M., Lauter, K.E., Song, Y.: Secure outsourced matrix computation and application to neural networks. In: CCS 2018. pp. 1209–1222. ACM (2018)
26. Liu, H., Wang, X., Yang, K., Yu, Y.: The hardness of LPN over any integer ring and field for PCG applications. IACR Cryptol. ePrint Arch. p. 712 (2022), https://eprint.iacr.org/2022/712
27. Liu, H., Xing, C., Yuan, C., Zou, T.: Dishonest majority multiparty computation over matrix rings. IACR Cryptol. ePrint Arch. p. 1912 (2023), https://eprint.iacr.org/2023/1912
28. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious neural network predictions via minionn transformations. In: ACM CCS 2017. pp. 619–631. ACM (2017)
29. Mohassel, P., Rindal, P.: Aby^3: A mixed protocol framework for machine learning. In: ACM CCS 2018. pp. 35–52. ACM (2018)
30. Mohassel, P., Zhang, Y.: Secureml: A system for scalable privacy-preserving machine learning. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 19–38. IEEE Computer Society (2017)
31. Orsini, E., Smart, N.P., Vercauteren, F.: Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In: CT-RSA 2020. LNCS, vol. 12006, pp. 254–283. Springer (2020)
32. Peter Rindal, L.R.: libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe
33. Rachuri, R., Scholl, P.: Le mans: Dynamic and fluid MPC for dishonest majority. In: CRYPTO 2022. LNCS, vol. 13507, pp. 719–749. Springer (2022)
34. Raghuraman, S., Rindal, P., Tanguy, T.: Expand-convolute codes for pseudorandom correlation generators from LPN. In: CRYPTO 2023. LNCS, vol. 14084, pp. 602–632. Springer (2023)
35. Schoppmann, P., Gascón, A., Reichert, L., Raykova, M.: Distributed vector-ole: Improved constructions and implementation. In: ACM CCS 2019. pp. 1055–1072. ACM (2019)
36. Weng, C., Yang, K., Katz, J., Wang, X.: Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1074–1091. IEEE (2021)

The Concrete Security of Two-Party Computation: Simple Definitions, and Tight Proofs for PSI and OPRFs

M. Bellare[1], R. Ranjan[1](✉), D. Riepel[2], and A. Aldakheel[3]

[1] Department of Computer Science and Engineering, University of California San Diego, San Diego, USA
{mbellare,riranjan}@ucsd.edu
[2] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
[3] Center of Excellence for Secure Computing, King Abdulaziz City for Science and Technology (KACST), Riyadh, Saudi Arabia
amaldakheel@kacst.edu.sa

Abstract. This paper initiates a concrete-security treatment of two-party secure computation. The first step is to propose, as target, a simple, indistinguishability-based definition that we call InI. This could be considered a poor choice if it were weaker than standard simulation-based definitions, but it is not; we show that for functionalities satisfying a condition called invertibility, that we define and show is met by functionalities of practical interest like PSI and its variants, the two definitions are equivalent. Based on this, we move forward to study the concrete security of a canonical OPRF-based construction of PSI, giving a tight proof of InI security of the constructed PSI protocol based on the security of the OPRF. This leads us to the concrete security of OPRFs, where we show how different DH-style assumptions on the underlying group yield proofs of different degrees of tightness, including some that are tight, for the well-known and efficient 2H-DH OPRF, and thus for the corresponding DH PSI protocol. We then give a new PSI protocol, called salted-DH PSI, that is as efficient as DH-PSI, yet enjoys tighter proofs.

1 Introduction

The first wave of research on secure two-party computation (2PC) [56] asked whether this magical-sounding goal could even be achieved. The focus being feasibility rather than efficiency, results were given in an asymptotic security framework and reduction tightness was not a concern. We are now in a second wave, fueled by real-world applications, where the focus is efficient protocols for particular goals like PSI and OPRFs. We suggest that, in this second wave, we need results in a concrete security framework, and reductions as tight as possible, so that we can find and pick the protocols that give the best efficiency for a desired level of proven security.

K.-M. Chung and Y. Sasaki (Eds.): ASIACRYPT 2024, LNCS 15489, pp. 328–362, 2025.
https://doi.org/10.1007/978-981-96-0938-3_11

Contributions in brief. Towards the above, this paper initiates a concrete-security treatment of 2PC. It has two main parts:

1. Definitional framework. We give and target a new, simple and concrete-security friendly definition of security for 2PC that we call input indistinguishability (InI). As the name indicates, it is indistinguishability based. Yet, for functionalities satisfying a condition called invertibility, that we define and is met by functionalities of practical interest including PSI and friends, we show that InI is as strong as standard simulation-based definitions. Our definitional framework explicitly incorporates random oracles and surfaces some subtleties in this regard.
2. Results for PSI and OPRFs. We consider the concrete security of OPRF-based PSI [32], giving a tight proof of InI security of the constructed PSI protocol based on the security of the starting OPRF. This motivates studying the concrete security of OPRFs, where we show how different DH-style assumptions on the underlying group yield proofs of different degrees of tightness, including some that are tight, for the well-known and efficient 2H-DH OPRF [36], and thus for DH-PSI, the PSI protocol based on 2H-DH. We follow this with a new protocol, salted DH PSI, for which we give tighter proofs. Salted DH PSI is essentially as efficient as DH PSI, showing how concrete security can be improved through protocol changes.

1.1 Setting the Stage

The asymptotic setting. Provable security [16,30] began in an *asymptotic framework* inherited from computational complexity theory. To show that a scheme Π meets a target notion of security T assuming that an underlying problem P (e.g. CDH) is hard, one gives a reduction that takes an adversary A and builds an adversary A' such that:

If A is PPT and has non-negligible advantage (success probability) $\epsilon_A(\cdot)$ in violating T-security of Π, then A' is also PPT and has non-negligible advantage $\epsilon_{A'}(\cdot)$ in breaking P.

The advantages here are functions of a security parameter k, and such a function is "negligible" if it goes to zero faster than the reciprocal of any polynomial. Such results help build theoretical foundations but give implementors no explicit way to pick the security parameter to guarantee a desired level (e.g. 256 bits) of security.

The concrete setting. In the *concrete framework* [9], one continues to give a reduction that takes an adversary A and builds an adversary A', but now additionally specifying a function $\mathbf{B}$, called the *bound*, such that:

If A has running time t, resources $\mathbf{R}$ and advantage $\mathbf{Adv}_{\Pi}^{\mathrm{T}}(A)$ in violating T-security of Π, then A' has running time about t and advantage ϵ' in breaking P such that $\mathbf{Adv}_{\Pi}^{\mathrm{T}}(A) \leq \mathbf{B}(\epsilon', \mathbf{R})$.

The advantages here are real numbers, not functions. There is no explicit security parameter. Resources include the number of queries to various oracles in the game defining security. A reduction is *tight* if $\mathbf{B}(\epsilon', \mathbf{R}) = c \cdot \epsilon'$ for some small constant c. A typical example of a non-tight reduction is $\mathbf{B}(\epsilon', \mathbf{R}) = q \cdot \epsilon'$ where $q \in \mathbf{R}$ is the number of queries of A to some oracle. Now if an implementor wants to ensure $\mathbf{Adv}^{\mathrm{T}}_{\Pi}(A) \leq \epsilon$ for some choices of $t, \epsilon, \mathbf{R}$, they can use the bound to determine ϵ' such that $\epsilon \leq \mathbf{B}(\epsilon', \mathbf{R})$ and then use t, ϵ' to make a choice of group or elliptic curve in which to work. The tighter the reduction, the smaller is ϵ' and thus the size of the curve, and the more efficient is the implemented scheme.

For example, suppose Π_1, Π_2 are protocols for some goal (say PSI), both proven secure under the CDH assumption over an underlying elliptic curve group, the first using six modular exponentiations in the group and second only three. Is Π_2 the one to prefer and implement? Not necessarily. The right comparison is at the same level of concrete security for both, say 128-bits. To provide this, say that, due to different degrees of tightness in the proofs for the two protocols, we need a 256-bit curve $\mathbb{G}_{256}$ for Π_1 and a 384-bit curve $\mathbb{G}_{384}$ for Π_2. Then (since exponentiation is cubic-time) Π_2 is $(3/6) \cdot (384/256)^3 = 1.6875$ times *more* expensive than Π_1, despite using *fewer* exponentiations in the group, making Π_1 the sounder choice.

Thus, beyond allowing sound choices of parameters, the concrete framework leads to new questions, such as to seek tighter reductions for existing protocols or to seek new protocols which allow tight reductions. These kinds of questions (which we will pursue for 2PC) are invisible in the asymptotic setting.

Concrete security is not new. In provable-security for symmetric cryptography, it is the norm, and it is widely employed in public-key cryptography and authenticated key-exchange.

<u>Definitional complexity.</u> Our intent is to facilitate and provide concrete security assessments and improvements for 2PC. The first step is simple, "concrete-security-friendly" definitions. We start with a broad definitional classification aimed at saying what this means.

Having fixed a target notion T and scheme or protocol Π, our discussions of security above assumed a simple definitional format in which the advantage $\mathbf{Adv}^{\mathrm{T}}_{\Pi}(A)$ is associated to just the adversary A, as is true for basic notions like UF-CMA (signatures), PRF-security or indistinguishability of encryptions. We will call these single-quantifier definitions since the security requirement is

$$\forall A \; : \mathbf{Adv}^{\mathrm{T}}_{\Pi}(A) \text{ is low.}$$

In a simulation-based definition, however, the advantage $\mathbf{Adv}^{\mathrm{T}}_{\Pi,\mathsf{S}}(A)$ is now relative to a simulator S. The requirement now being

$$\forall A \, \exists \mathsf{S} \; : \mathbf{Adv}^{\mathrm{T}}_{\Pi,\mathsf{S}}(A) \text{ is low} \quad \text{or} \quad \exists \mathsf{S} \, \forall A \; : \mathbf{Adv}^{\mathrm{T}}_{\Pi,\mathsf{S}}(A) \text{ is low,}$$

we refer to these as double-quantifier definitions. But this double-quantifier structure does not fit the above-discussed format and complicates concrete-security assessments.

To elaborate, double-quantifier definitions do not preclude giving concrete-security results, and there are some in the literature. (Examples include [55, Theorem 4] and [36, Theorem 1].) However it is not clear (at least to us) how to use such results to pick parameters to guarantee a desired level of security. One issue is that we would expect $\mathbf{Adv}^{\mathrm{T}}_{\mathsf{\Pi},\mathsf{S}}(A)$ to be lower for simulators with higher running time, raising the question of how to interpret this advantage and also making parameter choice depend on simulator running time. Also complexity grows when (as happens in the first just-cited result), one simulator is defined in terms of another, making it hard to work in a modular way.

Concrete-security friendliness is not the only benefit of single-quantifier definitions. Another is attack-friendliness. To give an attack, we need only give an adversary with high advantage. In a double-quantifier definition, we would have to prove that the advantage is high relative to *all* simulators. Our InI definition in particular facilitates cryptanalysis of 2PC protocols, a topic largely unexplored.

2PC. Recall that the setting of secure two-party computation (2PC) considers parties $1, 2$ (also called client and server, respectively) having inputs x_1, x_2 respectively. A 2-party protocol $\mathsf{\Pi}$, to securely compute a functionality F, allows the parties to interact so that at the end they have outputs y_1, y_2 respectively, where $(y_1, y_2) \leftarrow \mathsf{F}(x_1, x_2)$. Yet, neither party should learn more about the other party's input than disclosed by the output they obtain.

This area has traditionally used double-quantifier definitions in an asymptotic framework. But today the quest is efficient protocols for goals (functionalities) of interest in applications, where (for reasons given above) a concrete framework is crucial. Leading the way, concrete-security results for single-quantifier definitions have been given for garbling schemes [7]. However there are many protocols, for goals including PSI, OPRF and their variants, which target efficiency without concrete security. We aim to fill this gap.

1.2 Our Definitional Framework

The highlight of our framework below will be a definition for 2PC security, called InI, that is (1) single-quantifier and thus both concrete-security friendly and attack-friendly, yet (2) usually no less powerful than a standard (double-quantifier) simulation-based definition.

SIM and InI. We want to say (in a concrete framework) what it means for a protocol $\mathsf{\Pi}$ to securely compute a 2PC functionality F. The classical paradigm for 2PC definitions is simulation [19,43], so we start there, giving a concretely-rendered definition, called SIM. It defines an advantage $\mathbf{Adv}^{\mathrm{sim}}_{\mathsf{F},\mathsf{\Pi},\mathsf{S}}(A)$ for an adversary A relative to a simulator S. As above, this is a double-quantifier definition but represents an important baseline, in terms of strength and history, that we want to respect.

Alongside, we give a simple, single-quantifier definition that we call *input indistinguishability* (InI). Let $\mathsf{F}(x_1, x_2)[i]$ denote the output given by the functionality to party i. Suppose party 1 is the "honest" one, meaning the one whose privacy we aim to protect. Party 2, as the adversary A, supplies a pair of inputs

$$\begin{tikzcd}[column sep=huge, row sep=large]
\text{SIM-np} \arrow[r, "\textbf{Th. 1}"] & \text{SIM} \arrow[d, shift right=2, "\textbf{Th. 2}"'] \\
& \text{InI} \arrow[u, shift right=2, "/" marking, "\textbf{Th. 3}"'] \arrow[ul, thick, "{\textbf{Th. 4} \text{ for invertible functionalities}}"]
\end{tikzcd}$$

Fig. 1. Relations between the InI, SIM and SIM-np notions of security for a 2PC protocol Π for a functionality F. Arrows are implications and the barred arrow is a separation.

$x_{1,0}, x_{1,1}$ for party 1 and a single input x_2 for itself such that the outputs $y_0 = \mathsf{F}(x_{1,0}, x_2)[2]$ and $y_1 = \mathsf{F}(x_{1,1}, x_2)[2]$ for itself are the same. A random challenge bit b is chosen, and now A, given its view (transcript and coins) of the execution of protocol Π on inputs $x_{1,b}, x_2$, outputs a guess b'. InI asks that its advantage $\mathbf{Adv}^{\mathrm{InI}}_{\mathsf{F},\Pi}(A) = 2\Pr[b = b'] - 1$ is small. A formal definition is in Sect. 3.

Relating SIM and InI. We propose to use InI in concrete-security results and parameter choices. As the reader may note, this would be a poor choice if InI is weaker (provides less security) than SIM, but we show that, for functionalities of practical interest, this is not the case and in fact the two are equivalent. To elaborate, Theorem 4 says that InI implies SIM (that is, any Π that is InI-secure is also SIM-secure) as long as the target functionality F satisfies a condition we define called *invertibility*. The latter (continuing to assume party 1 is the honest one) asks that, given x_2, y_2, it is possible to efficiently find an input x_1 for party 1 satisfying $\mathsf{F}(x_1, x_2)[2] = y_2$, assuming of course such an x_1 exists. In the other direction, Theorem 2 says that SIM always implies InI.

In Sect. 3.3, we show that the functionality for PSI is invertible, as are variants of it like for threshold-PSI and cardinality-PSI [26]. We also show, in [12], invertibility for Oblivious Transfer (OT) [52] and the Secure Inferencing functionality of [39]. So for all these we may safely focus on InI, reassured that it is qualitatively just as strong as SIM.

We clarify and caution that it is not the case that InI and SIM are equivalent for *all* functionalities. Indeed, in Theorem 3 we give a counterexample, meaning a (non-invertible) F and a protocol Π such that Π is SIM-secure, but not InI secure, for F. However, the F, Π here are contrived and artificial. Our experience is that natural functionalities of practical interest tend to be invertible and thus enjoy the equivalence of InI and SIM guaranteed by Theorems 2, 4.

ROM incorporation and subtleties. The Random Oracle (RO) Model [13] is extensively employed for practical 2PC but, while used in proofs, the RO is sometimes absent in the definitions. Our definitions in contrast explicitly and flexibly incorporate random oracles. Protocols name a space from which their desired RO is then drawn in games defining security. In the default SIM notion, the RO is programmable by the simulator. We also give a non-programmable

RO version SIM-np. InI has the attractive, and further simplifying feature that the RO is inherently non-programmable. (There is no simulator to program it).

Attending carefully to formalizing the ROM usage in these definitions brought to light a subtle issue. RO queries could be made not only by the protocol and adversary, but also by the functionality. We show, by example, that allowing the simulator to program the answers to functionality RO queries is problematic and can lead to clearly insecure protocols having a proof of security. Our SIM definition addresses this, answering functionality queries via an honestly chosen random function that is then given as oracle to the simulator, who can use it, or not, as it likes, in answering other RO queries. (See Sect. 3.1 for details).

<u>Full relations picture.</u> With that, Fig. 1 summarizes the full set of relations between the notions. An arrow $\mathrm{X} \rightarrow \mathrm{Y}$ is an implication, meaning any Π that is X-secure for F is also Y-secure for F. A barred arrow $\mathrm{X} \not\rightarrow \mathrm{Y}$ is a separation, meaning there exist F, Π such that Π is X-secure for F but not Y-secure for F. For invertible F, we note that Theorem 4 actually shows InI $\rightarrow$ SIM-np, which implies InI $\rightarrow$ SIM because Theorem 1 says that SIM-np $\rightarrow$ SIM.

<u>The setting.</u> Our definitions and results are in the semi-honest (also called honest-but-curious) setting, where the parties aim to learn each other's inputs but are assumed to follow the protocol. While we want to eventually treat the malicious case, there are several reasons to start with the semi-honest one. The first is pedagogic; as our work shows, the semi-honest case is hardly trivial, and jumping to the malicious case without a solid foundation for the semi-honest one felt to us premature and unsound. The second reason is that many works in the literature [20,21,25,40,41,50,51] give protocols for the semi-honest setting, and understanding their concrete security is important for practical reasons. Pragmatic concerns too justify this setting. The gain is efficiency; malicious-secure protocols are typically more expensive, which may curtail adoption. Meanwhile, with regard to security, in practice there are forces external to the cryptography that deter malicious behavior. Parties are often corporations who are subject to laws and bound by contracts with other parties. Use of subverted (malicious) code risks discovery and exposure. Add to this that the protocol functionality is already giving these parties the information they want, and malicious behavior emerges as both low reward and high risk.

1.3 Concrete-Security Results for 2PC Protocols

We give some general results, and then focus on PSI and OPRFs.

<u>Many executions versus one.</u> In practice we expect that a protocol Π is executed many times, on different inputs. Our definitions accordingly allow the adversary to obtain as many execution transcripts as it likes via queries to an oracle Run. In the concrete setting we are interested in how adversary advantage degrades as a function of the number q_{rn} of queries to Run. Theorem 5 confirms that the hybrid argument works as expected to show that the advantage $\epsilon_{\Pi}^{\mathrm{InI}}(q_{\mathrm{rn}})$ for q_{rn} queries is at most q_{rn} times the advantage $\epsilon_{\Pi}^{\mathrm{InI}}(1)$ for one query.

In an asymptotic setting, the question would end here, but concretely, it is more of a starting point, raising the question of whether we can, for particular protocols, avoid the q_{rn} factor loss, meaning show that $\epsilon_{\Pi}^{\mathrm{InI}}(q_{\mathrm{rn}}) \approx \epsilon_{\Pi}^{\mathrm{InI}}(1)$. The following will show (amongst other things) that the answer is yes.

PSI FROM OPRFS, TIGHTLY. In Private Set Intersection (PSI) [26], the inputs x_1, x_2 are sets and the functionality $\mathsf{F}^{\mathrm{psi}}$ returns their intersection $x_1 \cap x_2$ to the client and nothing to the server. PSI is used for privacy-respecting solutions in the following domains: ad conversion [34,35], contact discovery [44], password or credential monitoring [3,33,42,54], genomics [4], proximity testing [47], relationship discovery in social networks [45] and detection of sexual misconduct [53]. These applications motivate PSI protocols with tight proofs.

Towards this, we focus on one simple, canonical way to achieve PSI suggested by Hazay and Lindell [32], where the PSI protocol Π^{psi} is built from a protocol Π^{oprf} for an Oblivious Pseudo-Random Function (OPRF) [25,38,46]. Recall that in the latter, the server has a (secret) key K for a (regular) PRF Q, the client has an input x and the protocol ends with them holding ε and $\mathsf{Q}(K, x)$, respectively. Simulation-based (hence double-quantifier) definitions of security for OPRFs are given in [55]. We give instead single-quantifier (non-simulation-based) definitions. For client security (honest party 1), it is simply InI. For server security (honest party 2) we give a simple pseudo-randomness definition that we call OPRF-PR. Under these assumptions, Theorem 6 shows client and server InI security of Π^{psi}. The reductions are all tight. This is true regardless of the number q_{rn} of Π^{psi}-executions (formally, RUN queries of the adversary), meaning the bounds do not have the multiplicative q_{rn} factor loss of the hybrid argument of Theorem 5. Theorem 6 also separately shows correctness of Π^{psi}, based on the PRF-security of Q. (The actual result is more general.)

Having thus stepped *tightly* from OPRFs to PSI, we turn to studying the concrete security of the former.

BOUNDS FOR 2H-DH OPRF AND DH-PSI. OPRFs have applications beyond PSI [22,23,36,37], making their concrete security of interest in its own right. 2H-DH (Two-Hash Diffie-Hellman) [36] is the de-facto standard OPRF and thus the natural candidate to study.

Jarecki, Kiayias and Krawczyk [36, Theorem 1] prove that 2H-DH achieves a simulation-based (UC) definition in the ROM, assuming hardness of the One-More Gap Diffie-Hellman (OM-Gap-DH) problem. This is a strong assumption, giving the adversary a CDH oracle in the One-More style [10,17] as well as a DDH oracle in the Gap style [48]. Their result is semi-concrete (a bound is given but the runtimes of the simulator and constructed adversaries are not), and the bound is not tight.

We revisit the 2H-DH OPRF and evaluate security under our (single quantifier) definitions, namely InI for client (party 1) and OPRF-PR for server (party 2), as needed for our application to PSI above. Theorem 9 shows client InI-security unconditionally and with a good bound. Our discussion focuses on OPRF-PR. We consider a variety of choices for the starting (assumed hard)

Problem P	$\mathbf{B}(\epsilon', \{q_{\rm ro}, q_{\rm n}\})$	$\mathbf{B}(\epsilon', \{q_{\rm ro}, q_{\rm rn}\})$	
	2H-DH OPRF	DH-PSI	Salted DH-PSI
CDH	$q_{\rm ro}^2 q_{\rm n} \cdot \epsilon'$	$q_{\rm ro}^2 q_{\rm rn} \cdot \epsilon'$	$q_{\rm ro} \cdot \epsilon'$
V-CDH	$q_{\rm ro} q_{\rm n} \cdot \epsilon'$	$q_{\rm ro} q_{\rm rn} \cdot \epsilon'$	ϵ'
CDH-MUC	$q_{\rm ro} \cdot \epsilon'$	$q_{\rm ro} \cdot \epsilon'$	$q_{\rm ro} \cdot \epsilon'$
V-CDH-MUC	ϵ'	ϵ'	ϵ'
DDH	ϵ'	ϵ'	ϵ'

Fig. 2. Our results for the 2H-DH OPRF and the DH-PSI and Salted DH-PSI protocols. For different choices of the assumed-hard problem P, the 2nd column shows the bound $\mathbf{B}(\epsilon', \{q_{\rm ro}, q_{\rm n}\})$ on the oprf-pr advantage of an adversary A for the 2H-DH OPRF, while the 3rd and 4th columns show the bound $\mathbf{B}(\epsilon', \{q_{\rm ro}, q_{\rm rn}\})$ on the ini-advantage of an adversary A for the DH-PSI and Salted DH-PSI protocols, respectively, in all cases as a function of the advantage $\epsilon' = \mathbf{Adv}_{\mathbb{G}}^{\rm P}(A')$ of the constructed adversary A' in solving problem P in group $\mathbb{G}$. In the first case, $q_{\rm ro}, q_{\rm n}$ are the number of queries A makes to its random and New oracles, respectively, and in the other cases, $q_{\rm ro}, q_{\rm rn}$ are the number of queries A makes to its random and Run oracles, respectively.

problem P in the group $\mathbb{G}$. What we consider interesting is that we can prove security under *all* these assumptions, but with *different tightness*.

The results are given in Theorem 10 and summarized in the second column of Fig. 2. It considers an OPRF-PR adversary A making $q_{\rm ro}$ queries to its random oracle and performing $q_{\rm n}$ executions (formally, queries to an oracle called New) of the OPRF protocol. Column 2 of the table then shows (approximate) bounds on the oprf-pr advantage of A as a function of $q_{\rm ro}, q_{\rm n}$ and the advantage ϵ' of a constructed adversary A' in solving problem P in group $\mathbb{G}$.

Row 1 of the table says that we can prove security already assuming hardness of only the (plain) CDH problem. But we incur a substantial factor loss in the bound. Now we consider strengthening the assumption. First, we give the adversary a limited DDH oracle. The resulting assumption, which we call V-CDH for verifiable CDH, is weaker than either Strong-CDH [1] or Gap-CDH [48]. Row 2 shows that the factor loss in the bound drops. Second, for both CDH and V-CDH, we move from the single-user setting to the one of multiple users with corruptions. Rows 2, 3 show further drops in the bound. Finally (row 5) we give a *tight* reduction from DDH. We refer to Sect. 2 for formal definitions of the computational problems and the relations between them.

Now, let DH-PSI denote the above-discussed PSI protocol when the OPRF is set to the 2H-DH one. Then, combining the above with Theorem 6 gives bounds on the server InI security of DH-PSI as shown in the 3rd column of Fig. 2.

Salted DH-PSI. Concrete security raises new questions invisible in the asymptotic setting, in this case whether there is a different protocol, ideally as efficient as DH-PSI, yet with bounds better than shown for the latter in Fig. 2. We show that the answer is yes, giving in Sect. 7 what we call the salted DH-PSI protocol. The bounds, as per Theorem 11 and summarized in the last column of Fig. 2, are improved under the CDH and V-CDH assumptions and maintained under the others. The salting technique we use originates in PSS [14], a modification of the RSA-FDH signature scheme which improved the bound, for UF-CMA under the RSA assumption, from loose to tight.

1.4 Discussion and Further Related Work

An indistinguishability-style definition for garbling schemes was given in [8], and one for multi-party computation in [2]. Our InI definition was inspired by, and generalizes, an indistinguishability-based definition for threshold-PSI from [5]. InI and SIM for 2PC can be seen as analogues of witness-indistinguishability [24] and zero-knowledge [31], respectively, for proof systems. Another domain in which both indistinguishability-style and simulation-style definitions have been given, related and used is functional encryption (FE) [11,18,49].

What we call single-quantifier and double-quantifier definitions are sometimes referred to as game-based and simulation-based, respectively. However games are a descriptive language and our SIM definition is also written as a game, so to avoid confusion we are using a different terminology that we feel highlights the essential difference, namely the quantifier structure.

There is a divide, in the cryptographic community, between those who speak and use the language of UC [19], and those who don't. A consequence has been to exclude a certain, and more applied part of our community, from 2PC research. Part of the intent of our work is to bridge this gap. With InI and concrete security, we have cast 2PC in a language and style similar to that used in practice-oriented work on conventional primitives like encryption, signatures and authenticated key exchange, primitives that have in particular seen a large quantity of work on proof tightness. The hope is to draw this segment of the community into 2PC to likewise explore and improve proof tightness.

In writing our definitions, we have aimed for precision, and attention to detail, at a level that to us is beyond the norm for the area. This is in part a response to our experience (admittedly perhaps due to our lack of expertise) of struggling to understand, and finding ambiguous, some definitions we try to read in the literature. A price paid is notation. Our work could (rightly) be critiqued as notationally heavy, but we believe the notation is central to greater precision and reduced ambiguity, and hope that, after some exposure, it ceases to be a significant barrier for a reader.

2 Preliminaries

Notation. If $\boldsymbol{w}$ is a vector then $|\boldsymbol{w}|$ is its length (the number of coordinates) and $\boldsymbol{w}[i]$ is its i-th coordinate. The empty (length zero) vector is denoted ε. We say that $\boldsymbol{w}$ is an n-vector if $|\boldsymbol{w}| = n$. We let $\mathsf{V2S}(\boldsymbol{w}) = \{\boldsymbol{w}[1], \ldots, \boldsymbol{w}[|\boldsymbol{w}|]\}$ be the set of elements of vector $\boldsymbol{w}$. Likewise, if S is a set, then $\boldsymbol{w} \leftarrow \mathsf{S2V}(S)$ puts its elements into a vector in some canonical order, say lexicographic. We write $\boldsymbol{w} \leftarrow\$ \mathsf{S2V}(S)$ to say that the ordering is random, meaning the entries of $\boldsymbol{w}$ are a random permutation of the elements of S. We say $\boldsymbol{w}$ is a vector over S if $\mathsf{V2S}(\boldsymbol{w}) \subseteq S$. By S^* we denote the set of all finite-length vectors over S.

Strings are identified with vectors over $\{0,1\}$, so that ε denotes the empty string, $\{0,1\}^*$ denotes the set of all finite-length strings, $|Z|$ denotes the length of a string Z and $Z[i]$ denotes its i-th bit. By $x \| y$ we denote the concatenation of strings x, y. If x, y are equal-length strings then $x \oplus y$ denotes their bitwise xor.

If X is a finite set, then $|X|$ denotes its size and $x \leftarrow\!\!\$\; X$ denote picking x uniformly at random from X. By $\mathcal{P}(X)$ we denote the power set of set X, meaning the set of all subsets of X. For integers $a \leq b$ we let $[a..b]$ be shorthand for $\{a, \ldots, b\}$. We use $1, 0$ to indicate the booleans "true" and "false" respectively, and $[[B]]$ returns 1 if boolean expression B is true and 0 otherwise. We use $\bot$ (bot) as a special symbol to denote rejection, and it is assumed to not be in $\{0,1\}^*$.

We let $\mathbb{G}^* = \mathbb{G} \setminus \{1\}$ be the set of non-identity elements of a group $\mathbb{G}$. By $\langle g \rangle$ we denote the set of all powers of $g \in \mathbb{G}$, so writing $\mathbb{G} = \langle g \rangle$ indicates that g is a generator of $\mathbb{G}$. In that case, $\mathrm{dlog}_{\mathbb{G},g}(A) \in \mathbb{Z}_p$ is the discrete logarithm of $A \in \mathbb{G}$ to base g, where p is the order of $\mathbb{G}$.

ORACLE SPACES AND RANDOM ORACLES. In the random oracle model [13], the domain and range of the random oracle can depend on the scheme. (The latter term here includes protocols and functionalities.) Accordingly, we let a scheme S specify a set OS (or $\mathsf{S.OS}$ if disambiguation is needed) of functions, called the *oracle space*. The game will then pick a function $\mathsf{H} \leftarrow\!\!\$\; \mathsf{OS}$ at random and provide as random oracle a procedure RO that when queried with X returns $\mathsf{H}(X)$. Finally, if OS is absent or empty, one is in the standard model directly.

ALGORITHMS. Functions (we will not consider uncomputable ones) are identified with deterministic algorithms. If OS is an oracle space (i.e. a set of functions) then we write $A\colon [\mathsf{OS}] \times D_1 \times \cdots \times D_n \to R$ to mean that A is an algorithm taking as oracle a function $\mathsf{H} \in \mathsf{OS}$ and taking inputs $x_1, \ldots, x_n$ with $x_i \in D_i$ for $i \in [1..n]$, to return an output $y \leftarrow\!\!\$\; A[\mathsf{H}](x_1, \ldots, x_n) \in R$. We let $\mathbf{Out}(A[\mathsf{H}](x_1, \ldots, x_n))$ denote the set of all possible outputs of A on the given inputs. Running time is worst case, which for an algorithm with access to an oracle means across all possible replies from the oracle. If we want to make A's coins (random choices) explicit we may see it as a deterministic algorithm $A\colon [\mathsf{OS}] \times D_1 \times \cdots \times D_n \times \Omega \to R$ so that $y \leftarrow\!\!\$\; A[\mathsf{H}](x_1, \ldots, x_n)$ is shorthand for picking $\omega \leftarrow\!\!\$\; \Omega$ and returning $y \leftarrow A[\mathsf{H}](x_1, \ldots, x_n; \omega)$. Omitting OS and the H argument return us to the standard model.

GAMES. We use the code-based game-playing framework of [15]. A game G specifies an INITIALIZE procedure, further procedures (also called oracles) and a FINALIZE procedure. In the ROM [13], which we use throughout, the random oracle appears as a game procedure RO. When game G is executed with adversary A, first INITIALIZE executes and what it returns is the input to A. Then A runs and can call oracles other than INITIALIZE, FINALIZE. When A halts, its output is the input to FINALIZE, and the output of the latter is the game output. By $\Pr[\mathrm{G}(A) \Rightarrow y]$ we denote the probability that the execution of game G with adversary A results in the game output being y, and write just $\Pr[\mathrm{G}(A)]$ for $\Pr[\mathrm{G}(A) \Rightarrow 1]$.

Different games may have procedures (oracles) with the same names, and if we need to disambiguate, we may write G.O to refer to oracle O of game G. In game pseudocode, integer variables, set variables, boolean variables and string variables are assumed initialized, respectively, to 0, the empty set $\emptyset$, the boolean 0 and $\perp$. Adversaries in games are always assumed to be domain-respecting, meaning if a query they provide is expected to fall in some scheme-associated set, then it does. The running time of an adversary by convention is the execution time of the game with the adversary, so that the time taken by oracles to respond to adversary queries is included. We write $\mathrm{Q}^{\mathrm{O}}(A)$ to denote the number of queries made to oracle O in the execution of the game with A. Note that by convention, again, both queries made directly by A and those made by scheme algorithms are included. In particular, $\mathrm{Q}^{\mathrm{RO}}(A)$ includes the queries made by scheme algorithms either explicitly to RO or instead directly to the function H underlying RO, in the execution of the game with A. We say that adversary A_2 (playing a game G_2) has the same query profile as adversary A_1 (playing a game G_1) if the games provide oracles of the same names (even if not same behavior), and the number of queries to each of these oracles is the same for both adversaries.

```
Game $\mathbf{G}^{\mathrm{prf}}_{\mathsf{Q}}$

INITIALIZE:
  $\mathsf{H} \leftarrow\$ \mathsf{Q.OS}$ ; $c \leftarrow\$ \{0,1\}$

NEW:
  $i \leftarrow i+1$ ; $k_i \leftarrow\$ \mathsf{Q.Keys}$

CH$(i', x)$:
  If not $(i' \le i)$ then return $\perp$
  If $\mathrm{T}[i', x] = \perp$ then
    If $c = 1$ then $\mathrm{T}[i', x] \leftarrow \mathsf{Q}[\mathrm{RO}](k_{i'}, x)$
    Else $\mathrm{T}[i', x] \leftarrow\$ \mathsf{R}$
  Return $\mathrm{T}[i', x]$

RO$(X)$:
  Return $\mathsf{H}(X)$

FINALIZE$(c')$:
  Return $[[c = c']]$
```

```
$\mathsf{2HDH}[\mathsf{H}](k, x)$:
  $Y \leftarrow \mathsf{H}(1, x)^k$   // $Y \in \mathbb{G}$
  $y \leftarrow \mathsf{H}(2, g^k, x, Y)$   // $y \in \{0,1\}^\ell$
  Return $y$

$\mathsf{F}^{\mathrm{psi}}_U(S_1, S_2)$:   // $S_1, S_2 \subseteq U$
  $I \leftarrow S_1 \cap S_2$
  Return $((I, |S_2|), |S_1|)$

$\mathsf{F}^{\mathrm{oprf}}_{\mathsf{Q}}[\mathsf{H}](\boldsymbol{x}, k)$:   // $\mathsf{V2S}(\boldsymbol{x}) \subseteq \mathsf{Q.D}$ and $k \in \mathsf{Q.Keys}$
  For $j = 1, \ldots, |\boldsymbol{x}|$ do
    $\boldsymbol{y}[j] \leftarrow \mathsf{Q}[\mathsf{H}](k, \boldsymbol{x}[j])$
  Return $(\boldsymbol{y}, |\boldsymbol{x}|)$
```

Fig. 3. Left: PRF game for function family Q. **Right:** On the top is the $\mathsf{2HDH}$ function family associated to group $\mathbb{G} = \langle g \rangle$ and integer ℓ, and, below it, the PSI functionality over universe U. At the bottom is the OPRF functionality associated to PRF Q.

<u>SECURITY OF FUNCTION FAMILIES.</u> A family of functions $\mathsf{Q}\colon [\mathsf{OS}] \times \mathsf{Keys} \times \mathsf{D} \to \mathsf{R}$ takes a key $K \in \mathsf{Keys}$ and input $X \in \mathsf{D}$ and, with oracle access to $\mathsf{H} \in \mathsf{OS}$, returns an output $Y \leftarrow \mathsf{Q}[\mathsf{H}](K, X)$. For emphasis or disambiguation, we may write $\mathsf{Q.OS}, \mathsf{Q.Keys}, \mathsf{Q.D}, \mathsf{Q.R}$ for the different subcomponents of Q.

A security metric for Q that we will use is PRF security [29] in the multi-user setting [6]. The prf (pseudorandom function) advantage of adversary A_{prf}

is defined as $\mathbf{Adv}^{\mathrm{prf}}_{\mathsf{Q}}(A_{\mathrm{prf}}) = 2\Pr[\mathbf{G}^{\mathrm{prf}}_{\mathsf{Q}}(A_{\mathrm{prf}})] - 1$ where the game is on the left in Fig. 3.

As an example, the top right of Fig. 3 shows the 2H-DH function family $\mathsf{2HDH}\colon [\mathsf{OS}] \times \mathbb{Z}_p \times \{0,1\}^* \to \{0,1\}^\ell$ underlying the 2H-DH OPRF [36]. It is associated to a group $\mathbb{G} = \langle g \rangle$ of prime order p generated by $g \in \mathbb{G}$, and an integer $\ell \geq 1$. Here OS is the set of all functions H such that $\mathsf{H}(1,\cdot)\colon \{0,1\}^* \to \mathbb{G}$ and $\mathsf{H}(2,\cdot,\cdot,\cdot)\colon \mathbb{G} \times \{0,1\}^* \times \mathbb{G} \to \{0,1\}^\ell$. This function family conceptually uses two random oracles $\mathsf{H}(1,\cdot), \mathsf{H}(2,\cdot,\cdot,\cdot)$ that are packaged into one to respect our formalism. The following says that $\mathsf{2HDH}$ is PRF-secure in the ROM.

Proposition 1. *Let $\mathbb{G} = \langle g \rangle$ be a group of prime order p, and $\ell \geq 1$ an integer. Let $\mathsf{2HDH}$ be the associated 2H-DH family of functions as per Fig. 3. Let A_{prf} be an adversary playing game $\mathbf{G}^{\mathrm{prf}}_{\mathsf{2HDH}}$. Then*

$$\mathbf{Adv}^{\mathrm{prf}}_{\mathsf{2HDH}}(A_{\mathrm{prf}}) \leq \frac{(\mathrm{Q}^{\mathrm{RO}}(A_{\mathrm{prf}}) + \mathrm{Q}^{\textsc{New}}(A_{\mathrm{prf}})) \cdot \mathrm{Q}^{\textsc{New}}(A_{\mathrm{prf}})}{p} .$$

We omit a formal proof, but the intuition is that, when the challenge bit is 1, outputs of the challenge oracle are distributed uniformly in R as long as a certain "bad" event does not happen, the event being either a collision in keys across New queries, or the random oracle being queried on $g^{k_{i'}}$ for a $k_{i'}$ picked by New. Thus it suffices to bound the probability of this bad event.

In Sect. 4, we show that server-side security of any OPRF implies PRF security of the family of functions underlying the OPRF.

3 2PC Definitional Framework

We give our core definitions of syntax and security in a concrete setting, and then turn to relations between definitions. We see the party identities as $1, 2$ with 1 being the "client" and 2 being the "server".

3.1 Core Definitions

Functionalities. A (two-party) functionality describes the function that the parties want to compute. Formally, it is an algorithm $\mathsf{F}\colon [\mathsf{OS}] \times \mathsf{D}_1 \times \mathsf{D}_2 \to \mathsf{R}_1 \times \mathsf{R}_2$. The functionalities of practical interest that we want to treat are deterministic, so for simplicity we restrict attention in this work to deterministic F, and this is assumed moving forward. We leave treatment of randomized functionalities to future work. Now, to explain, given as oracle $\mathsf{H} \in \mathsf{OS}$, and inputs $x_1 \in \mathsf{D}_1$ and $x_2 \in \mathsf{D}_2$ of parties 1,2 respectively, the functionality returns outputs $y_1 \in \mathsf{R}_1$ and $y_2 \in \mathsf{R}_2$ for parties 1,2, respectively, via $(y_1, y_2) \leftarrow \mathsf{F}[\mathsf{H}](x_1, x_2)$.

Allowing F to have access to a random oracle is important to capture some OPRFs. As per our vector notation, for $i \in \{1, 2\}$ we may write $\mathsf{F}[\mathsf{H}](x_1, x_2)[i]$ for the i-th component of the 2-vector $\mathsf{F}[\mathsf{H}](x_1, x_2)$.

```
$\underline{\mathsf{F}^{\mathrm{psi}}_{U}(S_1, S_2)\text{:}}$   // $S_1, S_2 \subseteq U$
1 $I \leftarrow S_1 \cap S_2$
2 Return $((I, |S_2|), |S_1|)$
```

```
$\underline{\mathsf{F}^{\mathrm{oprf}}_{\mathsf{Q}}[\mathsf{H}](\boldsymbol{x}, k)\text{:}}$   // $\mathsf{V2S}(\boldsymbol{x}) \subseteq \mathsf{Q.D}$ and $k \in \mathsf{Q.Keys}$
1 For $j = 1, \ldots, |\boldsymbol{x}|$ do
2     $\boldsymbol{y}[j] \leftarrow \mathsf{Q}[\mathsf{H}](k, \boldsymbol{x}[j])$
3 Return $(\boldsymbol{y}, |\boldsymbol{x}|)$
```

Fig. 4. Left: PSI functionality over universe U. **Right:** OPRF functionality associated to PRF Q.

PSI AND OPRF FUNCTIONALITIES. The right side of Fig. 4 shows two examples. First, the PSI functionality $\mathsf{F}^{\mathrm{psi}}_{U} : \mathcal{P}(U) \times \mathcal{P}(U) \to (\mathcal{P}(U) \times \mathbb{N}) \times \mathbb{N}$ is associated to a set U called the universe. This functionality does not use a random oracle. Party $i \in \{1, 2\}$ has input a set $S_i \subseteq U$. The intersection I of the two sets is returned to party 1, and both parties are also given set-size information because protocols tend to leak it.

Second, let $\mathsf{Q}\colon [\mathsf{OS}] \times \mathsf{Keys} \times \mathsf{D} \to \mathsf{R}$ be a family of functions. We associate to it the OPRF functionality $\mathsf{F}^{\mathrm{oprf}}_{\mathsf{Q}} : [\mathsf{OS}] \times \mathsf{D}^* \times \mathsf{Keys} \to \mathsf{R}^* \times \mathbb{N}$. The input of the server (party 2) is a PRF key $k \in \mathsf{Keys}$. The input of the client (party 1) is vector $\boldsymbol{x}$ over D. The functionality computes a corresponding vector $\boldsymbol{y}$, over R, of outputs under $\mathsf{Q}[\mathsf{H}](k, \cdot)$, that goes to the client. The server gets the length of $\boldsymbol{x}$ since protocols tend to leak it. In particular if $\mathsf{Q} = \mathsf{2HDH}$ is the $\mathsf{2HDH}$ PRF of Fig. 3 then $\mathsf{F}^{\mathrm{oprf}}_{\mathsf{Q}}$ is the 2H-DH OPRF functionality, protocols for which we will analyze. Note our definition extends the usual ones by allowing the client input to be a vector over D rather than a single point in D.

PROTOCOLS. Party $i \in \{1, 2\}$ has input x_i. The parties now use an interactive protocol to interact towards computing outputs for some target functionality. But what exactly (meaning, mathematically or definitionally) is a protocol? In UC [19] and Goldreich's textbooks [27,28], it is a pair of interactive TMs. In some parts of the literature (including Lindell's tutorial [43]) it is not formalized at all. We will give a usable yet rigorous formalization of a protocol as an algorithm that takes a current state and an incoming message to return an updated state and outgoing message.

Thus, formally, a *protocol* Π is an algorithm $\Pi\colon [\mathsf{OS}] \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$. It may be randomized, and $\Pi\colon [\mathsf{OS}] \times \{0,1\}^* \times \{0,1\}^* \times \Omega \to \{0,1\}^* \times \{0,1\}^*$ denotes the underlying deterministic algorithm with Ω the set of coins. As a function of its current state $st \in \{0,1\}^*$, a received message $m_{\mathrm{in}} \in \{0,1\}^*$ and its coins $\omega \in \Omega$, a party computes its outgoing message m_{out} as well as, for itself, an updated state st, written $(st, m_{\mathrm{out}}) \leftarrow \Pi[\mathsf{H}](st, m_{\mathrm{in}}; \omega)$. As usual, we write $(st, m_{\mathrm{out}}) \leftarrow\!\!\$\; \Pi[\mathsf{H}](st, m_{\mathrm{in}})$ for picking $\omega \leftarrow\!\!\$\; \Omega$ and letting $(st, m_{\mathrm{out}}) \leftarrow \Pi[\mathsf{H}](st, m_{\mathrm{in}}; \omega)$. A party's state records its input as st.in, its output as st.out and its decision to accept or reject as $st.\mathrm{dec} \in \{1, 0\}$. The interaction consists of $\mathsf{nr} \in \mathbb{N}$ moves (also called rounds). The convention is that party 1 sends the first message.

EXECUTION TRACES. A protocol may be (honestly) executed on inputs x_1, x_2, coins $\omega_1, \omega_2 \in \Omega$ for the parties and access to an oracle $\mathsf{H} \in \mathsf{OS}$ to generate an execution trace $(\tau, st_1, st_2) \leftarrow \mathbf{XT}_{\Pi}[\mathsf{H}](x_1, x_2; \omega_1, \omega_2)$. Here τ is a transcript of the interaction, which is the sequence of messages exchanged, and st_1, st_2 are the final states of the parties. In detail:

```
$\underline{\mathbf{XT}_{\Pi}[\mathsf{H}](x_1, x_2; \omega_1, \omega_2)}$
$st_1.\mathrm{in} \leftarrow x_1$ ; $st_2.\mathrm{in} \leftarrow x_2$ ; $m_0 \leftarrow \varepsilon$ ; $i \leftarrow 1$
For $j = 1, \ldots, \mathsf{nr}$ do
    $(st_i, m_j) \leftarrow \Pi[\mathsf{H}](st_i, m_{j-1}; \omega_i)$ ; $i \leftarrow 3 - i$
$\tau \leftarrow (m_1, \ldots, m_{\mathsf{nr}})$ ; Return $(\tau, st_1, st_2)$
```

As indicated above, the outputs and decisions can be recovered from the final states of the parties.

```
Game $\underline{\mathbf{G}^{\mathrm{corr}}_{\mathsf{F},\Pi}}$

INITIALIZE:
1 $\mathsf{H} \leftarrow\$ \mathsf{OS}$

RUN$(x_1, x_2)$:
2 $\omega_1, \omega_2 \leftarrow\$ \Omega$ ; $(\tau, st_1, st_2) \leftarrow \mathbf{XT}_{\Pi}[\mathsf{H}](x_1, x_2; \omega_1, \omega_2)$
3 $(y_1, y_2) \leftarrow \mathsf{F}[\mathsf{H}](x_1, x_2)$
4 If $(st_1.\mathrm{dec} = 0)$ or $(st_2.\mathrm{dec} = 0)$ then $\mathsf{win} \leftarrow 1$
5 If $(st_1.\mathrm{out} \neq y_1)$ or $(st_2.\mathrm{out} \neq y_2)$ then $\mathsf{win} \leftarrow 1$
6 Return $\mathsf{win}$

RO$(X)$:
7 Return $\mathsf{H}(X)$

FINALIZE:
8 Return $\mathsf{win}$
```

Fig. 5. Game assessing correctness of protocol Π for functionality F.

CORRECTNESS. Correctness asks that an honest execution of a protocol computes the target functionality. This is straightforward enough to define for perfect correctness, but we need a clear definition of imperfect correctness that in particular allows quantifying correctness failure in protocols where it depends on computational assumptions. Accordingly, we treat correctness in detail, using a game.

Let $\mathsf{F}\colon [\mathsf{OS}] \times \mathsf{D}_1 \times \mathsf{D}_2 \rightarrow \mathsf{R}_1 \times \mathsf{R}_2$ be a functionality and Π a protocol. We assume for simplicity that the functionality and protocol have the same oracle space, which is wlog. Define the correctness advantage of adversary A_{corr} as $\mathbf{Adv}^{\mathrm{corr}}_{\mathsf{F},\Pi}(A_{\mathrm{corr}}) = \Pr[\mathbf{G}^{\mathrm{corr}}_{\mathsf{F},\Pi}(A_{\mathrm{corr}})]$ where the game is in Fig. 5. Here the adversary can run the protocol on inputs (x_1, x_2) of its choice by calling oracle RUN. It wins if either the parties reject or their outputs do not match those of the functionality. Note that multiple calls to RUN are allowed. We say Π is perfectly correct for F if $\mathbf{Adv}^{\mathrm{corr}}_{\mathsf{F},\Pi}(A_{\mathrm{corr}}) = 0$ for all A_{corr}, regardless of the running time and number of oracle queries of A_{corr}. But having defined this advantage function allows us to make clear and precise statements about imperfect correctness.

This will allow us to see how the correctness advantage grows with the number of oracle queries in PSI protocols where correctness depends on computational assumptions.

Security. We will be considering security in the *semi-honest* or *honest-but-curious* model where it is assumed that the corrupt party does not deviate from the protocol but, at the end, given its view (conversation transcript and its own coins) tries to find information about the other party's input. By convention, we will refer to this other party as the honest one. We start with our new, single-quantifier indistinguishability-style definition that we call *input indistinguishability* (InI), and then give double-quantifier, simulation-style definitions SIM, SIM-np.

In our definitions, an adversary triggers a protocol execution, on inputs of its choice, via a query to an oracle Run. We allow multiple queries to Run, to capture the real-life expectation of multiple executions of the protocol on different inputs. This allows us to measure (and then reduce) the degradation of security as a function of the number of Run calls.

For all the following definitions, we let $\mathsf{F}\colon [\mathsf{OS}] \times \mathsf{D}_1 \times \mathsf{D}_2 \to \mathsf{R}_1 \times \mathsf{R}_2$ be the functionality. We let Π be a protocol for it with $\Pi\colon [\mathsf{OS}] \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$. We assume wlog that the RO spaces of F, Π are the same. (One can always work with an appropriate union of the two spaces if not.)

```
Game $\mathbf{G}^{\mathrm{ini}}_{\mathsf{F},\Pi,h}$

Initialize:
 1  $\mathsf{H} \leftarrow\$ \mathsf{OS}$ ; $b \leftarrow\$ \{0,1\}$

Run$(x_0, x_1, x)$:
 2  $x_{h,0} \leftarrow x_0$ ; $x_{h,1} \leftarrow x_1$ ; $x_{3-h,0} \leftarrow x$ ; $x_{3-h,1} \leftarrow x$
 3  $(y_{1,0}, y_{2,0}) \leftarrow \mathsf{F}[\mathsf{H}](x_{1,0}, x_{2,0})$
 4  $(y_{1,1}, y_{2,1}) \leftarrow \mathsf{F}[\mathsf{H}](x_{1,1}, x_{2,1})$
 5  If $(y_{3-h,0} \neq y_{3-h,1})$ then return $\bot$
 6  $\omega_1, \omega_2 \leftarrow\$ \Omega$
 7  $(\tau, st_1, st_2) \leftarrow \mathbf{XT}_{\Pi}[\mathsf{H}](x_{1,b}, x_{2,b}; \omega_1, \omega_2)$
 8  Return $(\tau, \omega_{3-h})$

RO$(X)$:
 9  Return $\mathsf{H}(X)$

Finalize$(b')$:
10  Return $[[b' = b]]$
```

```
Games $\mathbf{G}^{\text{sim-np}}_{\mathsf{F},\Pi,\mathsf{S},h}$, $\mathbf{G}^{\mathrm{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}$

Initialize:
 1  $\mathsf{H} \leftarrow\$ \mathsf{OS}$ ; $b \leftarrow\$ \{0,1\}$ ; $st_{\mathsf{S}} \leftarrow \varepsilon$

Run$(x_1, x_2)$:
 2  $(y_1, y_2) \leftarrow\$ \mathsf{F}[\mathsf{H}](x_1, x_2)$
 3  If $b = 1$ then
 4      $\omega_1, \omega_2 \leftarrow\$ \Omega$
 5      $(\tau, st_1, st_2) \leftarrow \mathbf{XT}_{\Pi}[\mathsf{H}](x_1, x_2; \omega_1, \omega_2)$
 6  Else
 7      $(\tau, \omega_{3-h}, st_{\mathsf{S}}) \leftarrow\$ \mathsf{S}[\mathsf{H}](\underline{\mathrm{run}}, (x_{3-h}, y_{3-h}), st_{\mathsf{S}})$
 8  Return $(\tau, \omega_{3-h})$

RO$(X)$:
 9  $h \leftarrow \mathsf{H}(X)$
10  If $b = 0$ then                                   // Game $\mathbf{G}^{\mathrm{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}$
11      $(h, st_{\mathsf{S}}) \leftarrow\$ \mathsf{S}[\mathsf{H}](\underline{\mathrm{ro}}, X, st_{\mathsf{S}})$      // Game $\mathbf{G}^{\mathrm{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}$
12  Return $h$

Finalize$(b')$:
13  Return $[[b' = b]]$
```

Fig. 6. Left: Game defining InI security for protocol Π for functionality F, where $h \in \{1,2\}$ is the honest party. **Right:** Games defining SIM-np (lines 10–11 excluded) and SIM (lines 10–11 included) security for protocol Π for a functionality F, where $h \in \{1,2\}$ is the honest party and S is the simulator.

InI definition. Input-indistinguishability (InI) is defined via game $\mathbf{G}^{\mathrm{ini}}_{\mathsf{F},\Pi,h}$ in Fig. 6. The ini-advantage of an adversary A_{ini} is then defined by $\mathbf{Adv}^{\mathrm{ini}}_{\mathsf{F},\Pi,h}(A_{\mathrm{ini}})$

$= 2\Pr[\mathbf{G}^{\mathrm{ini}}_{\mathsf{F},\Pi,h}(A_{\mathrm{ini}})] - 1$. To explain, here $h \in \{1,2\}$ is the "honest" party, meaning the adversary is playing the role of party $3-h$ and trying to learn something about the honest party's input. The adversary can query Run with two choices $x_0, x_1 \in \mathsf{D}_h$ of inputs for the honest party and a single choice $x \in \mathsf{D}_{3-h}$ for the corrupt party. This results in two pairs of inputs for the functionality. At lines 3–4 the functionality is evaluated on both pairs. If the resulting outputs $y_{3-h,0}$ and $y_{3-h,1}$ for the corrupt party differ, then the game returns $\perp$ at line 5 to avoid trivial distinguishing. Else, the protocol is run with the honest-party input being determined by the challenge bit b from line 1, and the resulting conversation transcript and the corrupt party's coins ω_{3-h} are returned to the adversary. Multiple queries to Run are allowed.

Asymptotically we would say that Π is InI secure for F if for every PPT A_{ini} the function $\mathbf{Adv}^{\mathrm{ini}}_{\mathsf{F},\Pi,h}(A_{\mathrm{ini}})$ is negligible, which illustrates how this is a single-quantifier definition. As usual in the concrete setting there is no formal definition of being "secure;" we give only a formal metric of security and our results in this concrete setting will relate advantages.

SIM-np definition. Moving to our simulation-based definitions, we start with SIM-np, the non-programmable ROM one. It is given via game $\mathbf{G}^{\mathrm{sim\text{-}np}}_{\mathsf{F},\Pi,\mathsf{S},h}$ in Fig. 6, where S is an algorithm called the simulator. As before, the game is also parameterized by the identity $h \in \{1,2\}$ of the honest party, whose input the adversary A_{snp}, in the role of the corrupted party $3-h$, is trying to learn. Lines 10–11 are not present in this game. Line 1 picks a random challenge bit b. If $b = 1$ then we have the "real" game and if $b = 0$ the "ideal" game. The adversary can call Run, giving it inputs for both parties. In response it obtains a conversation transcript τ, and coins ω_{3-h} for the corrupted party. It can query this oracle as often as it wants. In the real game, the transcript and coins are determined by running the protocol, while in the ideal game, they are determined by the simulator. Queries to the random oracle RO are answered via $\mathsf{H} \in \mathsf{OS}$. The first argument to the simulator is a keyword indicating the role in which it is being run, and st_{S} is its state. The latter is initialized at line 1. After that, when the simulator runs (line 7) it takes its current state and returns an updated state. The state variable st_{S} is maintained by the game. The non-programmability of the RO is in the fact that the RO oracle simply responds via H and the simulator gets access to the same H. We let $\mathbf{Adv}^{\mathrm{sim\text{-}np}}_{\mathsf{F},\Pi,\mathsf{S},h}(A_{\mathrm{snp}}) = 2\Pr[\mathbf{G}^{\mathrm{sim\text{-}np}}_{\mathsf{F},\Pi,\mathsf{S},h}(A_{\mathrm{snp}})] - 1$ be the advantage of an adversary A_{snp}.

In an asymptotic setting, we would say that Π is SIM-np secure for F and h if there is a PPT S such that for every PPT A_{snp} the function $\mathbf{Adv}^{\mathrm{sim\text{-}np}}_{\mathsf{F},\Pi,\mathsf{S},h}(A_{\mathrm{snp}})$ is negligible. This illustrates how this is a double-quantifier definition. As usual, in our concrete setting, theorems (e.g. Theorem 2) will relate advantages.

SIM definition. The programmable ROM version of our simulation-based definition of security, called SIM, is given via game $\mathbf{G}^{\mathrm{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}$ in Fig. 6. As before, $h \in \{1,2\}$ is the identity of the honest party and S is the simulator. Lines 10–11 (now included and the only change from SIM-np) represent the programming, allowing the simulator to determine the output of oracle RO. We continue to give the simulator access to an actual random oracle via H, which it can use or ignore as it wishes. As we will explain below, it is important for the meaningfulness of this definition that the functionality queries to the random oracle at line 2 are not programmed by, or even visible to, the simulator. We expect that st_{S} holds the current input-output table of the simulated random oracle, and whatever RO answers the simulator may need for the lines 7, 11 simulations, it can create and store in st_{S} if they do not already exist there. An adversary A_{sim} again has to find the correct value of the challenge bit b to win. We let $\mathbf{Adv}^{\mathrm{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}(A_{\mathrm{sim}}) = 2\Pr[\mathbf{G}^{\mathrm{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}(A_{\mathrm{sim}})] - 1$ be its advantage.

Again, in an asymptotic setting, we would say that Π is SIM-secure for F and h if there is a PPT S such that for every PPT A_{sim} the function $\mathbf{Adv}^{\mathrm{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}(A_{\mathrm{sim}})$ is negligible.

A SUBTLE POINT ABOUT SIM. At line 2 in game $\mathbf{G}^{\mathrm{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}$ (right panel of Fig. 6), RO queries of the functionality F, if any, are answered by an honest random function H. This may not be the first or obvious choice; why not have these also be answered by the simulator like the answers to other RO queries in this game? To explain, let us denote by SIM* the variant we have just mentioned, namely it is the same as SIM except that, at line 2, we replace $\mathsf{F}[\mathsf{H}](x_1, x_2)$ with $\mathsf{F}[\mathrm{RO}](x_1, x_2)$, so that, when $b = 0$, the RO queries of F are answered by the simulator at line 11. A self-contained and formal definition of SIM*, as well as a more precise and formal rendition of what follows, is in [12].

We claim that SIM* is an incorrect (unsound) definition for functionalities F that access the RO. (If F does not access RO, there is no difference between SIM and SIM*, and both are sound.) Specifically, our claim is that, if F can access the RO, then obviously insecure protocols can be shown secure under SIM*.

As an example, let $\mathsf{F} = \mathsf{F}^{\mathrm{oprf}}_{\mathsf{2HDH}}$ be the OPRF functionality (Fig. 4) associated to the 2H-DH PRF $\mathsf{2HDH}\colon [\mathsf{OS}] \times \mathbb{Z}_p \times \{0,1\}^* \rightarrow \{0,1\}^\ell$ (Fig. 3). Recall that here OS is the set of all functions H such that $\mathsf{H}(1,\cdot)\colon \{0,1\}^* \rightarrow \mathbb{G}$ and $\mathsf{H}(2,\cdot,\cdot,\cdot)\colon \mathbb{G} \times \{0,1\}^* \times \mathbb{G} \rightarrow \{0,1\}^\ell$. Suppose party 1 (client) has input $x \in \{0,1\}^*$ —formally, the 1-vector (x)— while party 2 (server) has input a key $k \in \mathbb{Z}_p$. Consider the following protocol Π: (1) Party 1 sends its entire input x to party 2 (2) party 2 computes $Y \leftarrow \mathsf{H}(1,x)^k$, sends (Y, g^k) to party 1, and outputs 1 as its own output, and finally (3) party 1 computes and outputs $y \leftarrow \mathsf{H}(2, g^k, x, Y)$.

This protocol should clearly be considered insecure for honest party $h = 1$ since from the conversation transcript an adversary learns the entire input x of party 1, which it cannot deduce given just the functionality output (namely 1) for the corrupted party (namely party 2). Yet, it is possible to design a successful simulator for Π under SIM*. Why? At line 2 on the right of Fig. 6, F would query $X = (1,x)$ to RO to compute $Y \leftarrow \mathsf{H}(1,x)^k$. But this query X is passed at line 11 to the simulator, who thus directly learns x. It can store x in its state, and can now easily produce the transcript τ at line 7. Namely, it knows the input k of the corrupted party and can thus compute $Y \leftarrow \mathsf{H}(1,x)^k$ and return $(x, (Y, g^k))$ as the transcript. So this protocol is SIM* secure despite being intuitively insecure. This anomaly goes away with SIM, where now the query $1, x$ made to H at line 2 is not visible to the simulator.

3.2 Relations Between Definitions

In this section we give the formal result statements corresponding to the relations shown in Fig. 1.

<u>SIM-np implies SIM.</u> The following confirms that SIM-np implies SIM, meaning the non-programmable ROM definition is stronger than the programmable one. The proof is simple and is omitted.

Theorem 1. [SIM-np $\Rightarrow$ SIM] *Let* F *be a functionality and* Π *a protocol for it. Let* $h \in \{1, 2\}$ *be the honest party. Given a simulator* S *defining* $\mathsf{S}[\cdot](\underline{\text{run}}, \cdot, \cdot)$*, extend it to also define* $\mathsf{S}[\cdot](\underline{\text{ro}}, \cdot, \cdot)$ *by* $\mathsf{S}[\mathsf{H}](\underline{\text{ro}}, X, st_{\mathsf{S}}) = (\mathsf{H}(X), st_{\mathsf{S}})$*. Then for any adversary* A *we have*

$$\mathbf{Adv}^{\text{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}(A) = \mathbf{Adv}^{\text{sim-np}}_{\mathsf{F},\Pi,\mathsf{S},h}(A)\ . \tag{1}$$

How does this statement show that SIM-np implies SIM? Assume Π is SIM-np-secure for F. Then there is a PPT SIM-np-simulator S such that $\mathbf{Adv}^{\text{sim-np}}_{\mathsf{F},\Pi,\mathsf{S},h}(A)$ is negligible for all PPT A. The extended S defined by the theorem is a PPT SIM-simulator, and (1) implies that $\mathbf{Adv}^{\text{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}(A)$ is also negligible for all PPT A, and Π is thus SIM-secure. In terms of reductions, (1) represents a trivial one which maps A to itself.

<u>SIM implies InI.</u> The following says that SIM always implies InI. That is, if Π is SIM secure for F then Π is also InI secure for F. The proof is in [12].

Theorem 2. [SIM $\Rightarrow$ InI] *Let* F *be a functionality and* Π *a protocol for it. Let* $h \in \{1, 2\}$ *be the honest party. Let* A_{ini} *be an adversary playing game* $\mathbf{G}^{\text{ini}}_{\mathsf{F},\Pi,h}$*. Then we can construct an adversary* A_{sim} *such that for all simulators* S *we have*

$$\mathbf{Adv}^{\text{ini}}_{\mathsf{F},\Pi,h}(A_{\text{ini}}) \le 2 \cdot \mathbf{Adv}^{\text{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}(A_{\text{sim}})\ . \tag{2}$$

Adversary A_{sim} *has the same query profile as* A_{ini} *and about the same running time.*

It may seem strange that (2) holds for *all* simulators. In particular, how does this show that SIM implies InI? The answer is that if Π is SIM-secure for F then there is a particular, PPT simulator S such that $\mathbf{Adv}^{\text{sim}}_{\mathsf{F},\Pi,\mathsf{S},h}(A_{\text{sim}})$ is negligible. Now by using S in (2) we can conclude that $\mathbf{Adv}^{\text{ind}}_{\mathsf{F},\Pi,h}(A_{\text{ini}})$ is also negligible, meaning Π is InI-secure for F.

<u>InI does not always imply SIM.</u> We now give a functionality F, and a protocol Π for it, such that Π is InI-secure but not SIM-secure. We assume for this the hardness of the discrete logarithm problem in a cyclic group $\mathbb{G}$. The formalization for the latter is via the DL game $\mathbf{G}^{\text{dl}}_{\mathbb{G},g,p}$ shown in the left panel of Fig. 7. It is associated to group $\mathbb{G} = \langle g \rangle$ of order p with generator $g \in \mathbb{G}$. The advantage of an adversary A_{dl} playing this game is given by $\mathbf{Adv}^{\text{dl}}_{\mathbb{G},g,p}(A_{\text{dl}}) = \Pr[\mathbf{G}^{\text{dl}}_{\mathbb{G},g,p}(A_{\text{dl}})]$. The proof of the following is in [12].

```
Game $\mathbf{G}^{\mathrm{dl}}_{\mathbb{G},g,p}$

INITIALIZE:
  $x \leftarrow\!\!\$\; \mathbb{Z}_p$ ; Return $g^x$

FINALIZE($z$):
  Return $[[z = x]]$
```

Party 1		**Party 2**
Input: x		Input: ε
	$\xrightarrow{x}$	
Output: ε		Output: g^x

Fig. 7. Left: Discrete log game for group $\mathbb{G} = \langle g \rangle$ of order p. **Right:** Protocol Π for Theorem 3.

Theorem 3 [*InI* $\not\Rightarrow$ SIM in general]**.** *Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of order p. Let $\mathsf{F} : \mathbb{Z}_p \times \{\varepsilon\} \to \{\varepsilon\} \times \mathbb{G}$ be the functionality defined by $\mathsf{F}(x, \varepsilon) = (\varepsilon, g^x)$ for all $x \in \mathbb{Z}_p$. Let Π be the protocol for F shown in Fig. 7. Then:*

1. *$\underline{\Pi \text{ is ini-secure for } \mathsf{F}}$: For any adversary A_{ini} playing game $\mathbf{G}^{\mathrm{ini}}_{\mathsf{F},\Pi,1}$, we have*

$$\mathbf{Adv}^{\mathrm{ini}}_{\mathsf{F},\Pi,1}(A_{\mathrm{ini}}) = 0 \, . \tag{3}$$

2. *$\underline{\Pi \text{ is sim-insecure for } \mathsf{F} \text{ assuming the DL problem is hard}}$: For all simulators S, there exist adversaries $A_{\mathrm{sim}}, A_{\mathrm{dl}}$, playing games $\mathbf{G}^{\mathrm{sim}}_{\mathsf{F},\Pi,\mathsf{S},1}$ and $\mathbf{G}^{\mathrm{dl}}_{\mathbb{G},g,p}$, respectively, such that*

$$\mathbf{Adv}^{\mathrm{sim}}_{\mathsf{F},\Pi,\mathsf{S},1}(A_{\mathrm{sim}}) = 1 - \mathbf{Adv}^{\mathrm{dl}}_{\mathbb{G},g,p}(A_{\mathrm{dl}}) \, . \tag{4}$$

Adversary A_{dl} has the same running time as an execution of S in its run *role, and A_{sim} runs in constant time.*

Why does (4) mean that Π is not SIM-secure? Let S be any PPT simulator. Then the Theorem gives PPT adversaries $A_{\mathrm{sim}}, A_{\mathrm{dl}}$ such that (4) holds. But assuming DL is hard, $\mathbf{Adv}^{\mathrm{dl}}_{\mathbb{G},g,p}(A_{\mathrm{dl}})$ is negligible, so the equation is saying that A_{sim} has a high (close to 1) advantage, which shows that Π is not SIM-secure for S. Since S was arbitrary, Π is not SIM-secure.

INVERTIBILITY. We define *invertibility* for functionalities with respect to the honest-party identity $h \in \{1, 2\}$. Let $\mathsf{F} : [\mathsf{OS}] \times \mathsf{D}_1 \times \mathsf{D}_2 \to \mathsf{R}_1 \times \mathsf{R}_2$ be a functionality. An algorithm $\mathrm{IA} : [\mathsf{OS}] \times \mathsf{D}_{3-h} \times \mathsf{R}_{3-h} \to \mathsf{D}_h$ is called an *inverter* for F and h if for all $\mathsf{H} \in \mathsf{OS}$ and all $(x_1, x_2) \in \mathsf{D}_1 \times \mathsf{D}_2$, the following always returns 1:

```
$(y_1, y_2) \leftarrow \mathsf{F}[\mathsf{H}](x_1, x_2)$   // Get functionality outputs
$x'_h \leftarrow\!\!\$\; \mathrm{IA}[\mathsf{H}](x_{3-h}, y_{3-h})$   // Resample an input for honest party
$x'_{3-h} \leftarrow x_{3-h}$   // Input unchanged for corrupt party
$(y'_1, y'_2) \leftarrow \mathsf{F}[\mathsf{H}](x'_1, x'_2)$   // Get new functionality outputs
Return $[[y'_{3-h} = y_{3-h}]]$   // Require corrupted-party output to be unchanged
```

Intuitively, consider an entity (this will be the simulator in our usage) who has an input x_{3-h} for the corrupted party. It also has an output y_{3-h} for the corrupted party, obtained from x_{3-h} and some (unknown to this entity) input x_h for the honest party. Invertibility asks that, given these, it is possible for our entity to efficiently find an input x'_h for the honest party that "explains" the output obtained by the corrupted party. It need not be that $x'_h = x_h$, and similarly need not be that $y'_h = y_h$.

In an asymptotic setting, we would say that a functionality F is *invertible* for h if there exists a PPT inverter IA for F and h. In our concrete setting, we will include the running time of IA in results.

InI implies SIM-np for invertible functionalities. Let F be a functionality that is invertible for $h \in \{1, 2\}$. The following says that any protocol that is InI secure for h is SIM-np secure (and thus by Theorem 1 also SIM secure) for h. The proof is in [12].

Theorem 4 [InI $\Rightarrow$ SIM-np for invertible functionalities]. *Let $h \in \{1, 2\}$ be the honest party. Let F be a functionality which is invertible for h, using inverter* IA. *Let Π be a protocol for F. Then there is a simulator S such that the following is true. Let $A_{\rm snp}$ be any adversary playing game $\mathbf{G}^{\text{sim-np}}_{\mathsf{F},\Pi,\mathsf{S},h}$. Then we can construct an adversary $A_{\rm ini}$ playing game $\mathbf{G}^{\rm ini}_{\mathsf{F},\Pi,h}$ such that*

$$\mathbf{Adv}^{\text{sim-np}}_{\mathsf{F},\Pi,\mathsf{S},h}(A_{\rm snp}) \le \mathbf{Adv}^{\rm ini}_{\mathsf{F},\Pi,h}(A_{\rm ini}) \; . \tag{5}$$

Adversary $A_{\rm ini}$ has the same query profile as $A_{\rm snp}$. Its running time is about that of $A_{\rm snp}$. The running time of S is that of Π plus the time for an execution of IA.

```
$\mathsf{F}^{\rm psi}_U(S_1, S_2)$:
  $I \leftarrow S_1 \cap S_2$
  Return $((I, |S_2|), |S_1|)$

$\mathrm{IA}^{\rm psi}_{U,2}(S_1, (I, s_2))$:
  $S'_2 \leftarrow I$ ; $Y \leftarrow S_1$
  While $(|S'_2| < s_2)$ do
      Pick some $d \in U \setminus Y$
      $Y \leftarrow Y \cup \{d\}$
      $S'_2 \leftarrow S'_2 \cup \{d\}$
  Return $S'_2$

$\mathrm{IA}^{\rm psi}_{U,1}(S_2, s_1)$:
  $S'_1 \leftarrow \emptyset$
  While $(|S'_1| < s_1)$ do
      Pick some $d \in U \setminus S'_1$
      $S'_1 \leftarrow S'_1 \cup \{d\}$
  Return $S'_1$
```

```
$\mathsf{F}^{\rm tpsi}_{U,t}(S_1, S_2)$:
  $I \leftarrow S_1 \cap S_2$
  If $|I| < t$ then return $((\bot, |S_2|), |S_1|)$
  Return $((I, |S_2|), |S_1|)$

$\mathrm{IA}^{\rm tpsi}_{U,2}(S_1, (I, s_2))$:
  If $I = \bot$ then
      $S'_2 \leftarrow \emptyset$
      While $(|S'_2| < s_2)$ do
          Pick some $d \in U \setminus S'_2$
          $S'_2 \leftarrow S'_2 \cup \{d\}$
  Else
      $S'_2 \leftarrow\!\$\; \mathrm{IA}^{\rm psi}_{U,2}(S_1, (I, s_2))$
  Return $S'_2$

$\mathrm{IA}^{\rm tpsi}_{U,1}(S_2, s_1)$:
  $S'_1 \leftarrow\!\$\; \mathrm{IA}^{\rm psi}_{U,1}(S_2, s_1)$
  Return $S'_1$
```

```
$\mathsf{F}^{\rm cpsi}_U(S_1, S_2)$:
  $I \leftarrow S_1 \cap S_2$
  Return $((|I|, |S_2|), |S_1|)$

$\mathrm{IA}^{\rm cpsi}_{U,2}(S_1, (n, s_2))$:
  Pick some $I \subseteq S_1$ with $|I| = n$
  $S'_2 \leftarrow\!\$\; \mathrm{IA}^{\rm psi}_{U,2}(S_1, (I, s_2))$
  Return $S'_2$

$\mathrm{IA}^{\rm cpsi}_{U,1}(S_2, s_1)$:
  $S'_1 \leftarrow\!\$\; \mathrm{IA}^{\rm psi}_{U,1}(S_2, s_1)$
  Return $S'_1$
```

Fig. 8. Left: PSI functionality and its inverters. **Middle:** tPSI functionality and its inverters. **Right:** cPSI functionality and its inverters.

3.3 Invertibility of PSI and Friends

Theorem 4 says that InI is just as strong as SIM-np as long as the functionality is invertible. Here we show that PSI [26], as well as a collection of PSI-related functionalities, are all invertible. This means that, for these functionalities, we can target InI without loss of security compared to simulation-based definitions, gaining in this way from the simplicity and concrete-security friendliness that the former offers compared to the latter. We show invertibility for some more functionalities in [12].

Proceeding, Fig. 8 shows three PSI-related functionalities that have arisen in the literature. Below each are inverters for it, first for honest party 2 and then for honest party 1, demonstrating invertibility of that functionality for both parties. The set U is the universe. We now discuss these in turn.

PSI. The PSI functionality $\mathsf{F}_U^{\mathrm{psi}} \colon \mathcal{P}(U) \times \mathcal{P}(U) \to (\mathcal{P}(U) \times \mathbb{N}) \times \mathbb{N}$ in the first panel is the same as in Fig. 4, repeated for clarity. Here $S_1, S_2 \subseteq U$.

The inverter for party 2 takes as input an input set S_1 for party 1 and an output (I, s_2) for party 1, where I is the intersection of S_1 with some (unknown to the inverter) set S_2 of party 2, and $s_2 = |S_2|$. The inverter aims to construct some (any) set S_2' of size s_2 such that $S_1 \cap S_2' = I$. It does this as shown.

The inverter for party 1 is easier. It gets an input set S_2 for party 2 and an output s_1 that is the size of some (unknown to the inverter) set S_1 of party 1. It aims to construct some (any) set S_1' of size s_1, done as shown.

Both inverters are linear time. They are thus efficient as required for invertibility.

Threshold PSI. The threshold-PSI (tPSI) functionality [26] $\mathsf{F}_{U,t}^{\mathrm{tpsi}} \colon \mathcal{P}(U) \times \mathcal{P}(U) \to ((\mathcal{P}(U) \cup \{\perp\}) \times \mathbb{N}) \times \mathbb{N}$ is parameterized, in addition to U, by an integer $t \geq 0$ which specifies the threshold that the cardinality of intersection of S_1 and S_2 must reach for the intersection to appear in the output of party 1. Clearly when $t = 0$, the tPSI functionality is same as the basic PSI functionality.

The inverter for party 2 takes input an input set S_1 for party 1 and an output (I, s_2) for party 1, where I is either the intersection of S_1 with some (unknown to the inverter) set S_2 of party 2 or is $\perp$, and $s_2 = |S_2|$. In the case that $I = \perp$, the inverter picks and returns some (any) set S_2' of size s_2. If $I \neq \perp$, it runs the PSI inverter. The inverter for party 1 is the same as for PSI. The inverters are again linear time.

Cardinality PSI. The cardinality-PSI (cPSI) functionality [26] $\mathsf{F}_U^{\mathrm{cpsi}} \colon \mathcal{P}(U) \times \mathcal{P}(U) \to (\mathbb{N} \times \mathbb{N}) \times \mathbb{N}$ provides the cardinality of the intersection, rather than the intersection itself, in the output for party 1.

The inverter for party 2 takes input an input set S_1 for party 1 and an output (n, s_2) for party 1, where n is the size of the intersection of S_1 with some (unknown to the inverter) set S_2 of party 2, and $s_2 = |S_2|$. The inverter aims to construct some (any) set S_2' of size s_2 such that $|S_1 \cap S_2'| = n$. It does this as shown. The inverter for party 1 is the same as for PSI, and as before the inverters are linear time.

3.4 General Composition Result

In practice, a 2PC protocol will be executed many times on different inputs. We want to prove that this is secure. To that end, we consider general composition and ask whether security for a single execution security implies security for multiple executions. As one might expect, a simple hybrid argument does work and the claim below formalizes just that. The proof is in [12].

Theorem 5. *Let* F *be a functionality. Let* $h \in \{1, 2\}$ *be the honest party. Let* Π *be a protocol for* F*. Let* $A_{\rm ini}$ *be an adversary playing game* $\mathbf{G}^{\rm ini}_{\mathsf{F},\Pi,h}$*. Then we can construct an adversary* $B_{\rm ini}$*, also playing game* $\mathbf{G}^{\rm ini}_{\mathsf{F},\Pi,h}$ *but making at most one* Run *query, such that*

$$\mathbf{Adv}^{\rm ini}_{\mathsf{F},\Pi,h}(A_{\rm ini}) \leq \mathrm{Q}^{\textsc{Run}}(A_{\rm ini}) \cdot \mathbf{Adv}^{\rm ini}_{\mathsf{F},\Pi,h}(B_{\rm ini}) \,. \tag{6}$$

Additionally $\mathrm{Q}^{\rm RO}(B_{\rm ini}) = \mathrm{Q}^{\rm RO}(A_{\rm ini})$ *and the running time of* $B_{\rm ini}$ *is about that of* $A_{\rm ini}$*.*

Asymptotically, this would end the question, but concretely it is more of a starting point, for it raises the question of showing security for multiple executions tightly, meaning with the same bound as for a single execution rather than with the linear degradation of the hybrid argument. In the following sections we will do this for OPRF and PSI protocols.

4 PSI from OPRFs

In this section, we evaluate the concrete security of the canonical OPRF-based PSI protocol of Hazay and Lindell [32]. To do this, we first give definitions for OPRFs.

Oblivious Pseudorandom Functions. Let $\mathsf{Q}\colon [\mathsf{OS}] \times \mathsf{Keys} \times \mathsf{D} \to \mathsf{R}$ be a family of functions. The OPRF functionality $\mathsf{F}^{\rm oprf}_{\mathsf{Q}} : [\mathsf{OS}] \times \mathsf{D}^* \times \mathsf{Keys} \to \mathsf{R}^* \times \mathbb{N}$ associated to Q was defined in Fig. 4. We say that protocol Π is an OPRF for Q if it computes the functionality $\mathsf{F}^{\rm oprf}_{\mathsf{Q}}$ with perfect correctness.

In an OPRF protocol, the server input is a secret key $k \in \mathsf{Keys}$. Conventionally, the client input would be a point in D, but we generalize this; in our setting the client input is a vector $\boldsymbol{x}$ over D. The client output is the vector $(\mathsf{Q}(k, \boldsymbol{x}[1]), \ldots, \mathsf{Q}(k, \boldsymbol{x}[|\boldsymbol{x}|]))$ and the server output is $|\boldsymbol{x}|$.

OPRF security. Let protocol Π be an OPRF for $\mathsf{Q}\colon [\mathsf{OS}] \times \mathsf{Keys} \times \mathsf{D} \to \mathsf{R}$. We separately define OPRF-security of Π for party 1 (client) and party 2 (server).

The client-security definition is simple, namely just InI-security as defined in Sect. 3. This says that the server cannot obtain information about the client input. This shows how we can leverage our definitional framework for OPRFs.

For server-security, we give a very simple definition of pseudorandomness that we call OPRF-PR. Namely, we take the game defining PRF security of function family Q in Fig. 3 and simply add an oracle that allows the adversary to

```
Game $\mathbf{G}^{\text{oprf-pr}}_{\Pi,\mathsf{Q}}$

INITIALIZE:
 1 $\mathsf{H} \leftarrow\!\!\$\ \mathsf{OS}$ ; $c \leftarrow\!\!\$\ \{0,1\}$

NEW:
 2 $i \leftarrow i+1$ ; $k_i \leftarrow\!\!\$\ \mathsf{Q.Keys}$

TR$(i', \boldsymbol{x})$:
 3 If not $(i' \leq i)$ then return $\bot$
 4 If $(\exists j : \mathrm{T}[i', \boldsymbol{x}[j]] \neq \bot)$ then return $\bot$
 5 $\omega_1, \omega_2 \leftarrow\!\!\$\ \Omega$
 6 $(\tau, st_1, st_2) \leftarrow \mathbf{XT}_{\Pi}[\mathrm{RO}](\boldsymbol{x}, k_{i'}; \omega_1, \omega_2)$
 7 $\boldsymbol{y} \leftarrow st_1.\mathrm{out}$
 8 For $j = 1, \ldots, |\boldsymbol{y}|$ do $\mathrm{T}[i', \boldsymbol{x}[j]] \leftarrow \boldsymbol{y}[j]$
 9 Return $(\boldsymbol{y}, \tau, \omega_1)$

CH$(i', x)$:
 10 If not $(i' \leq i)$ then return $\bot$
 11 If $\mathrm{T}[i', x] = \bot$ then
 12     If $c = 1$ then $\mathrm{T}[i', x] \leftarrow \mathsf{Q}[\mathsf{H}](k_{i'}, x)$
 13     Else $\mathrm{T}[i', x] \leftarrow\!\!\$\ \mathsf{R}$
 14 Return $\mathrm{T}[i', x]$

RO$(X)$:
 15 $h \leftarrow \mathsf{H}(X)$ ; Return $h$

FINALIZE$(c')$:
 16 Return $[[c = c']]$
```

Fig. 9. OPRF-PR game for pseudo-randomness (server side security) of an OPRF protocol Π. Here Q is the underlying PRF.

obtain transcripts of resulting game $\mathbf{G}^{\text{oprf-pr}}_{\Pi,\mathsf{Q}}$ is shown in Fig. 9. Challenge oracle CH is as in the PRF game in [12]. The transcript oracle TR takes a vector $\boldsymbol{x}$ of client inputs and (line 6) executes the protocol to obtain a conversation transcript and final states of the parties. From the final states, it extracts the party outputs, and uses the client outputs to update the table that stores the challenge function. Note that these entries are always the real ones as computed by the protocol, meaning, if $c = 0$, the challenge entries are random but the ones created by the transcript oracle are still real. The advantage of adversary A_{oprf} is $\mathbf{Adv}^{\text{oprf-pr}}_{\Pi,\mathsf{Q}}(A_{\text{oprf}}) = 2\Pr[\mathbf{G}^{\text{oprf-pr}}_{\Pi,\mathsf{Q}}(A_{\text{oprf}})] - 1$.

A definition of pseudorandomness for OPRFs is also given in [55], but it is simulation-based and thus double-quantifier. Our simpler definition is single-quantifier.

The following says that if Π is a OPRF-PR-secure OPRF for a function family Q, then the latter is PRF-secure. The proof is trivial and is omitted.

Proposition 2. *Let protocol Π be an OPRF for function family $\mathsf{Q}\colon [\mathsf{OS}] \times \mathsf{Keys} \times \mathsf{D} \to \mathsf{R}$. Let A_{prf} be an adversary playing game $\mathbf{G}^{\text{prf}}_{\mathsf{Q}}$. Then we can construct an adversary A_{oprf} playing game $\mathbf{G}^{\text{oprf-pr}}_{\Pi,\mathsf{Q}}$ such that*

$$\mathbf{Adv}^{\text{prf}}_{\mathsf{Q}}(A_{\text{prf}}) \leq \mathbf{Adv}^{\text{oprf-pr}}_{\Pi,\mathsf{Q}}(A_{\text{oprf}}) . \tag{7}$$

Adversary A_{oprf} has the same query profile and running time as A_{prf}, in particular making no TR *queries.*

PSI FROM OPRF. Now that we have security definitions for OPRFs, we analyze the InI security of the classic OPRF-based protocol from [32]. The protocol, which we denote Π^{psi}, is shown in Fig. 10. It is associated to a family of functions

$\mathsf{Q}\colon [\mathsf{OS}] \times \mathsf{Keys} \times U \to \mathsf{R}$ and an OPRF $\Pi^{\rm oprf}$ for Q. The random oracle $\mathsf{H} \in \mathsf{OS}$ is used by $\Pi^{\rm oprf}$ and Q, both of which are used by $\Pi^{\rm psi}$. The universe U is the domain of Q. The protocol $\Pi^{\rm psi}$ computes the functionality $\mathsf{F}^{\rm psi}_U$ defined in Fig. 4. The following says that OPRF tightly implies PSI, meaning there is a tight reduction from $\Pi^{\rm oprf}$ to $\Pi^{\rm psi}$. Note that PRF security of Q, as required by part 1, is not an extra assumption due to Proposition 2. The proof is in [12].

PSI Protocol $\Pi^{\rm psi}$

Based on: Function family $\mathsf{Q}\colon [\mathsf{OS}] \times \mathsf{Keys} \times U \to \mathsf{R}$ and OPRF $\Pi^{\rm oprf}$ for Q
Party 1 input: Set $S_1 \subseteq U$ of size s_1
Party 2 input: Set $S_2 \subseteq U$ of size s_2
Oracle: $\mathsf{H} \in \mathsf{OS}$ available to all parties.

1. Party 1 constructs its input vector $\boldsymbol{x} \leftarrow \mathsf{S2V}(S_1)$ for the OPRF protocol from the elements of its set S_1.
2. Party 2 picks a key $k \leftarrow\$ \mathsf{Keys}$ for the PRF Q.
3. The parties run the OPRF $\Pi^{\rm oprf}$ with the input of party 1 being $\boldsymbol{x}$ and that of party 2 being k. The output of party 1 is $\boldsymbol{y} \in \mathsf{R}^{|\boldsymbol{x}|}$, and that of party 2 is $|\boldsymbol{x}| = s_1$.
4. Party 2 lets $Z = \{\, \mathsf{Q}[\mathsf{H}](k, s) \ :\ s \in S_2 \,\}$ and $\boldsymbol{z} \leftarrow \mathsf{S2V}(Z)$. It sends $\boldsymbol{z}$ to party 1.
5. Party 1 constructs intersection set I as $I \leftarrow \{\, \boldsymbol{x}[i] \ :\ 1 \le i \le s_1 \text{ and } \boldsymbol{y}[i] \in \mathsf{V2S}(\boldsymbol{z}) \,\}$. It also computes $|\boldsymbol{z}| = s_2$.

Party 1 output: Set $I \subseteq U$ and size s_2 of S_2
Party 2 output: Size s_1 of S_1.

Fig. 10. PSI protocol $\Pi^{\rm psi}$ associated to function family Q and OPRF $\Pi^{\rm oprf}$ for Q.

Theorem 6. *Let $U \subseteq \{0,1\}^*$ be a set (the universe), and $\mathsf{F} = \mathsf{F}^{\rm psi}_U$ the associated PSI functionality. Let $\Pi^{\rm oprf}$ be an OPRF for a family of functions $\mathsf{Q}\colon [\mathsf{OS}] \times \mathsf{Keys} \times U \to \mathsf{R}$. Let $\Pi = \Pi^{\rm psi}$ be the PSI protocol built from Q and $\Pi^{\rm oprf}$ as in Fig. 10. Then:*

1. *Π is correct for F if Q is a PRF: Let $A_{\rm psi}$ be an adversary playing game $\mathbf{G}^{\rm corr}_{\mathsf{F},\Pi}$. Then we can construct an adversary $A_{\rm prf}$ such that*

$$\mathbf{Adv}^{\rm corr}_{\mathsf{F},\Pi}(A_{\rm psi}) \le \mathbf{Adv}^{\rm prf}_{\mathsf{Q}}(A_{\rm prf}) + \sum_{i=1}^{q} \frac{s_{i,1}s_{i,2}}{|\mathsf{R}|} \ . \tag{8}$$

Here $q = \mathrm{Q}^{\rm RUN}(A_{\rm psi})$ and $s_{i,j}$ is the upper bound on the size of party j in the i-th Run *query. Also $\mathrm{Q}^{\rm NEW}(A_{\rm prf}) = q$ and $\mathrm{Q}^{\rm CH}(A_{\rm prf}) = \sum_{i=1}^{q}(s_{i,1} + s_{i,2})$ and $\mathrm{Q}^{\rm RO}(A_{\rm prf}) = \mathrm{Q}^{\rm RO}(A_{\rm psi})$. The running time of $A_{\rm prf}$ is about that of $A_{\rm psi}$.*
2. *Π provides InI client security if $\Pi^{\rm oprf}$ does: Let $A_{\rm psi}$ be an adversary playing game $\mathbf{G}^{\rm ini}_{\mathsf{F},\Pi,1}$. Then we can construct an adversary $A_{\rm oprf}$ playing game $\mathbf{G}^{\rm ini}_{\mathsf{F}^{\rm oprf}_{\mathsf{Q}},\Pi^{\rm oprf},1}$ such that*

$$\mathbf{Adv}^{\rm ini}_{\mathsf{F},\Pi,1}(A_{\rm psi}) \le \mathbf{Adv}^{\rm ini}_{\mathsf{F}^{\rm oprf}_{\mathsf{Q}},\Pi^{\rm oprf},1}(A_{\rm oprf}) \ . \tag{9}$$

Adversary A_{oprf} makes the same number of Run *queries as A_{psi}, with the vector in each of length the (common) size of the two client sets in the corresponding query of A_{psi}. Also, $\mathrm{Q}^{\mathrm{RO}}(A_{\mathrm{oprf}}) \leq \mathrm{Q}^{\mathrm{RO}}(A_{\mathrm{psi}}) + \sum_{i=1}^{q} s_{i,2}$ where $q = \mathrm{Q}^{\mathrm{Run}}(A_{\mathrm{psi}})$ and $s_{i,2}$ is the upper bound on the size of party 2's set in the i-th* Run *query. The running time of A_{oprf} is about that of A_{psi}.*

3. *Π provides InI server security if Π^{oprf} is OPRF-PR secure: Let A_{psi} be an adversary playing game $\mathbf{G}^{\mathrm{ini}}_{\mathsf{F},\Pi,2}$. Then we can construct an adversary A_{oprf} playing game $\mathbf{G}^{\mathrm{oprf\text{-}pr}}_{\Pi^{\mathrm{oprf}},\mathsf{Q}}$ such that*

$$\mathbf{Adv}^{\mathrm{ini}}_{\mathsf{F},\Pi,2}(A_{\mathrm{psi}}) \leq 2 \cdot \mathbf{Adv}^{\mathrm{oprf\text{-}pr}}_{\Pi^{\mathrm{oprf}},\mathsf{Q}}(A_{\mathrm{oprf}}) \ . \tag{10}$$

Let $q = \mathrm{Q}^{\mathrm{Run}}(A_{\mathrm{psi}})$. Then $\mathrm{Q}^{\mathrm{New}}(A_{\mathrm{oprf}}) = \mathrm{Q}^{\mathrm{TR}}(A_{\mathrm{oprf}}) = q$ and $\mathrm{Q}^{\mathrm{CH}}(A_{\mathrm{oprf}}) \leq \sum_{i=1}^{q} s_{i,2}$ and $\mathrm{Q}^{\mathrm{RO}}(A_{\mathrm{oprf}}) = \mathrm{Q}^{\mathrm{RO}}(A_{\mathrm{psi}})$ where $s_{i,2}$ is an upper bound on the size of party 2's set(s) in the i-th Run *query of A_{psi}. The running time of A_{oprf} is about that of A_{psi}.*

The above tightly bounds the InI security of Π^{psi} via the OPRF security of Π^{oprf}. So if we can concretely bound the OPRF security of Π^{oprf}, we can pick parameters to use in practice for Π^{psi} to guarantee a desired level of security. Accordingly we now turn to proving security with concrete bounds for a canonical OPRF.

5 Computational Problems over the Group

We will show concrete OPRF-PR-security of the 2H-DH OPRF based on the hardness of a variety of different computational problems over the underlying group. In each case, we will give explicit bounds on the OPRF-PR-advantage as a function of the advantage in solving the group problem. These bounds will differ, and the intent is exactly to showcase how the choice of group problem affects the bound. In this section we define the relevant computational problems and give relations between them.

The problems. Let $\mathbb{G} = \langle g \rangle$ be a group of prime order p with generator g. The problems we consider are CDH, DDH, V-CDH, which are in the single-user setting, multi-user versions CDH-MU, V-CDH-MU, DDH-MU, and multi-user with corruptions versions CDH-MUC and V-CDH-MUC. All problems are defined via the games in Fig. 11. Throughout Fig. 11, writing game names next to an oracle means that only the named games include the oracle. If there is no annotation, all the games include that oracle. For $\mathrm{xx} \in \{\mathrm{cdh}, \mathrm{v\text{-}cdh}, \mathrm{cdh\text{-}mu}, \mathrm{v\text{-}cdh\text{-}mu}, \mathrm{cdh\text{-}muc}, \mathrm{v\text{-}cdh\text{-}muc}\}$ we define the advantage of an adversary A_{xx} by $\mathbf{Adv}^{\mathrm{xx}}_{\mathbb{G},g,p}(A_{\mathrm{xx}}) = \Pr[\mathbf{G}^{\mathrm{xx}}_{\mathbb{G},g,p}(A_{\mathrm{xx}})]$. For $\mathrm{xx} \in \{\mathrm{ddh}, \mathrm{ddh\text{-}mu}\}$ we define the advantage of an adversary A_{xx} by $\mathbf{Adv}^{\mathrm{xx}}_{\mathbb{G},g,p}(A_{\mathrm{xx}}) = 2\Pr[\mathbf{G}^{\mathrm{xx}}_{\mathbb{G},g,p}(A_{\mathrm{xx}})] - 1$.

Relations between problems. We will prove security of the 2H-DH OPRF directly under some assumptions and get bounds under others via relations between the assumptions. Figure 12 shows a diagram of the relations. Here

```
Games $\mathbf{G}^{\text{v-cdh}}_{\mathbb{G},g,p}$, $\mathbf{G}^{\text{cdh}}_{\mathbb{G},g,p}$

INITIALIZE:
  $k \leftarrow\!\!\$\ \mathbb{Z}_p$ ; $K \leftarrow g^k$ ; $B \leftarrow\!\!\$\ \mathbb{G}$
  Return $(K, B)$

DDHO($Z'$): // $\mathbf{G}^{\text{v-cdh}}_{\mathbb{G},g,p}$
  Return $[[Z' = B^k]]$

FINALIZE($T$):
  Return $[[Z = B^k]]$
```

```
Game $\mathbf{G}^{\text{ddh}}_{\mathbb{G},g,p}$

INITIALIZE:
  $k \leftarrow\!\!\$\ \mathbb{Z}_p$ ; $K \leftarrow g^k$ ; $B \leftarrow\!\!\$\ \mathbb{G}$
  $Z_1 \leftarrow B^k$ ; $Z_0 \leftarrow\!\!\$\ \mathbb{G} \setminus \{Z_1\}$ ; $c \leftarrow\!\!\$\ \{0,1\}$
  Return $(K, B, Z_c)$

FINALIZE($c'$):
  Return $[[c = c']]$
```

```
Games $\mathbf{G}^{\text{cdh-mu}}_{\mathbb{G},g,p}$, $\mathbf{G}^{\text{v-cdh-mu}}_{\mathbb{G},g,p}$, $\mathbf{G}^{\text{cdh-muc}}_{\mathbb{G},g,p}$, $\mathbf{G}^{\text{v-cdh-muc}}_{\mathbb{G},g,p}$

NEWKEY:
  $i \leftarrow i + 1$ ; $k_i \leftarrow\!\!\$\ \mathbb{Z}_p$ ; $K_i \leftarrow g^{k_i}$
  Return $K_i$

NEWBASE:
  $j \leftarrow j + 1$ ; $B_j \leftarrow\!\!\$\ \mathbb{G}$
  Return $B_j$

DDHO($i', j', Z'$): // $\mathbf{G}^{\text{v-cdh-mu}}_{\mathbb{G},g,p}$, $\mathbf{G}^{\text{v-cdh-muc}}_{\mathbb{G},g,p}$
  If not $(i' \leq i)$ or not $(j' \leq j)$ then return $\perp$
  Return $[[Z' = B_{j'}^{k_{i'}}]]$

CDHO($i', j'$): // $\mathbf{G}^{\text{v-cdh-muc}}_{\mathbb{G},g,p}$, $\mathbf{G}^{\text{cdh-muc}}_{\mathbb{G},g,p}$
  If not $(i' \leq i)$ or not $(j' \leq j)$ then return $\perp$
  $S \leftarrow S \cup \{(i', j')\}$ ; Return $B_{j'}^{k_{i'}}$

FINALIZE($i', j', Z$):
  If not $(i' \leq i)$ or not $(j' \leq j)$ then return 0
  If $(i', j') \in S$ then return 0
  Return $[[\, Z = B_{j'}^{k_{i'}} \,]]$
```

```
Game $\mathbf{G}^{\text{ddh-mu}}_{\mathbb{G},g,p}$

INITIALIZE:
  $c \leftarrow\!\!\$\ \{0,1\}$

NEWKEY:
  $i \leftarrow i + 1$ ; $k_i \leftarrow\!\!\$\ \mathbb{Z}_p$ ; $K_i \leftarrow g^{k_i}$
  Return $K_i$

NEWBASE:
  $j \leftarrow j + 1$ ; $B_j \leftarrow\!\!\$\ \mathbb{G}$
  Return $B_j$

CH($i', j'$):
  If not $(i' \leq i)$ or not $(j' \leq j)$ then return $\perp$
  If $\mathrm{T}[i', j'] = \perp$
    If $c = 1$ then $\mathrm{T}[i', j'] \leftarrow B_{j'}^{k_{i'}}$
    Else $\mathrm{T}[i', j'] \leftarrow\!\!\$\ \mathbb{G}$
  Return $\mathrm{T}[i', j']$

FINALIZE($c'$):
  Return $[[c = c']]$
```

Fig. 11. Here $\mathbb{G}$ is a group with prime order p and generator g. **Left:** Games for the CDH and V-CDH problems (top) and CDH-MU, V-CDH-MU, CDH-MUC and V-CDH-MUC problems (bottom). The DDHO oracle is only present in games $\mathbf{G}^{\text{v-cdh}}_{\mathbb{G},g,p}$, $\mathbf{G}^{\text{v-cdh-mu}}_{\mathbb{G},g,p}$ and $\mathbf{G}^{\text{v-cdh-muc}}_{\mathbb{G},g,p}$. The CDHO oracle is only present in games $\mathbf{G}^{\text{cdh-muc}}_{\mathbb{G},g,p}$ and $\mathbf{G}^{\text{v-cdh-muc}}_{\mathbb{G},g,p}$. **Right:** Game for the DDH problem (top) and the DDH-MU problem (bottom).

$$\begin{tikzcd}
 & \text{V-CDH-MUC} \arrow[r] \arrow[d, shift right] & \text{CDH-MUC} \arrow[d, shift right] \\
\text{DDH-MU} \arrow[ur] \arrow[r] \arrow[d, shift right] & \text{V-CDH-MU} \arrow[r] \arrow[u, shift right, "q_{\text{nk}} \cdot q_{\text{nb}}"'] \arrow[d, shift right] & \text{CDH-MU} \arrow[u, shift right, "q_{\text{nk}} \cdot q_{\text{nb}}"'] \arrow[d, shift right] \\
\text{DDH} \arrow[r] \arrow[u, shift right] & \text{V-CDH} \arrow[r] \arrow[u, shift right] & \text{CDH} \arrow[u, shift right]
\end{tikzcd}$$

Fig. 12. Diagram showing relations between the assumptions. The arrows represent implications. Here $q_{\text{nk}} = \mathrm{Q}^{\text{NEWKEY}}(A)$ and $q_{\text{nb}} = \mathrm{Q}^{\text{NEWBASE}}(A)$ where A is playing the game $\mathbf{G}^{\mathrm{P}}_{\mathbb{G},g,p}$ for $\mathrm{P} \in \{\text{V-CDH-MUC}, \text{CDH-MUC}\}$.

"$\mathrm{P}_1 \rightarrow \mathrm{P}_2$" is an implication, and means that, if P_1 is hard in group $\mathbb{G}$ then P_2 is also hard in $\mathbb{G}$. If an arrow is annotated with a value, for example $q_{\mathrm{nk}} \cdot q_{\mathrm{nb}}$, it means the reduction looses this factor. If there is no annotation, the reduction is tight. Some of the implications are trivial and easy to see, for example, V-CDH-MUC $\rightarrow$ CDH-MUC and CDH-MUC $\rightarrow$ CDH-MU. Standard re-randomization allows us to tightly obtain DDH $\rightarrow$ DDH-MU, CDH $\rightarrow$ CDH-MU and V-CDH $\rightarrow$ V-CDH-MU. The reductions V-CDH $\rightarrow$ V-CDH-MUC and CDH $\rightarrow$ CDH-MUC as well as DDH $\rightarrow$ V-CDH-MUC are more interesting. The first two are captured by the following theorem, whose proof is in [12].

Theorem 7. *Let $\mathbb{G} = \langle g \rangle$ be a group with prime order p. Let A_{v} and A be adversaries playing the $\mathbf{G}^{\text{v-cdh-muc}}_{\mathbb{G},g,p}$ game and the $\mathbf{G}^{\text{cdh-muc}}_{\mathbb{G},g,p}$ game, respectively. Then we can construct adversaries $A_{\text{v-cdh}}$ and A_{cdh} playing the $\mathbf{G}^{\text{v-cdh}}_{\mathbb{G},g,p}$ and $\mathbf{G}^{\mathrm{cdh}}_{\mathbb{G},g,p}$ games, respectively, such that*

$$\mathbf{Adv}^{\text{v-cdh-muc}}_{\mathbb{G},g,p}(A_{\mathrm{v}}) \leq \mathrm{Q}^{\textsc{NewKey}}(A_{\mathrm{v}}) \cdot \mathrm{Q}^{\textsc{NewBase}}(A_{\mathrm{v}}) \cdot \mathbf{Adv}^{\text{v-cdh}}_{\mathbb{G},g,p}(A_{\text{v-cdh}}) \ , \quad (11)$$

$$\mathbf{Adv}^{\text{cdh-muc}}_{\mathbb{G},g,p}(A) \leq \mathrm{Q}^{\textsc{NewKey}}(A) \cdot \mathrm{Q}^{\textsc{NewBase}}(A) \cdot \mathbf{Adv}^{\mathrm{cdh}}_{\mathbb{G},g,p}(A_{\mathrm{cdh}}) \ . \quad (12)$$

The running time of $A_{\text{v-cdh}}$ is about that of A_{v} plus the time for $(\mathrm{Q}^{\textsc{NewKey}}(A_{\mathrm{v}}) + \mathrm{Q}^{\textsc{NewBase}}(A_{\mathrm{v}}) + \mathrm{Q}^{\mathrm{CDHO}}(A_{\mathrm{v}}) + \mathrm{Q}^{\mathrm{DDHO}}(A_{\mathrm{v}}))$ group exponentiations, and the running time of A_{cdh} is about that of A plus the time for $(\mathrm{Q}^{\textsc{NewKey}}(A) + \mathrm{Q}^{\textsc{NewBase}}(A) + \mathrm{Q}^{\mathrm{CDHO}}(A))$ group exponentiations.

Curiously, we can show DDH $\rightarrow$ V-CDH-MUC with a tight reduction. This is captured by the following, whose proof is in [12].

Theorem 8. *Let $\mathbb{G} = \langle g \rangle$ be a group with prime order p. Let A_{v} be an adversary playing the $\mathbf{G}^{\text{v-cdh-muc}}_{\mathbb{G},g,p}$ game. Then we can construct an adversary A_{ddh} playing the $\mathbf{G}^{\mathrm{ddh}}_{\mathbb{G},g,p}$ game such that*

$$\mathbf{Adv}^{\text{v-cdh-muc}}_{\mathbb{G},g,p}(A_{\mathrm{v}}) \leq \mathbf{Adv}^{\mathrm{ddh}}_{\mathbb{G},g,p}(A_{\mathrm{ddh}}) + \frac{\mathrm{Q}^{\mathrm{DDHO}}(A_{\mathrm{v}}) + 1}{p} \ . \quad (13)$$

The running time of A_{ddh} is about that of A_{v} plus the time for at most $(\mathrm{Q}^{\textsc{NewKey}}(A_{\mathrm{v}}) + 2 \cdot \mathrm{Q}^{\textsc{NewBase}}(A_{\mathrm{v}}) + 2 \cdot \mathrm{Q}^{\mathrm{CDHO}}(A_{\mathrm{v}}) + 2 \cdot \mathrm{Q}^{\mathrm{DDHO}}(A_{\mathrm{v}}))$ group exponentiations.

6 Security of 2H-DH OPRF

We have shown (Theorem 6) that PSI can be built *tightly* from an OPRF. Now we turn to seeing how tightly we can build OPRFs based on algebraic assumptions. For this purpose we consider 2H-DH [36], a leading and very efficient OPRF. We will showcase how its security can be proven under different algebraic assumptions with different degrees of tightness. We note that the 2H-DH OPRF has many applications beyond PSI [22,23,36,37], making our results about it of independent interest.

2H-DH OPRF. We fix a group $\mathbb{G} = \langle g \rangle$ of prime order p with generator g. We also fix an integer $\ell \geq 1$. Recall that we associated to $\mathbb{G}, \ell$ the family of functions $\mathsf{2HDH}\colon [\mathsf{OS}] \times \mathbb{Z}_p \times \{0,1\}^* \to \{0,1\}^\ell$ specified in Sect. 2. It uses a random oracle $\mathsf{H} \in \mathsf{OS}$ which specifies two sub-functions: $\mathsf{H}(1,\cdot)\colon \{0,1\}^* \to \mathbb{G}$ and $\mathsf{H}(2,\cdot,\cdot,\cdot)\colon \mathbb{G} \times \mathbb{G} \times \{0,1\}^* \times \mathbb{G} \to \{0,1\}^\ell$. Succinctly, $\mathsf{2HDH}[\mathrm{RO}](k, x) = \mathsf{H}(2, g^k, x, \mathsf{H}(1, x)^k)$. The 2H-DH protocol is shown in Fig. 13. It realizes the functionality $\mathsf{F}^{\mathrm{oprf}}_{\mathsf{2HDH}}$ with perfect correctness. The client (party 1) has input a vector $\boldsymbol{x}$ over $\{0,1\}^*$, and the server has input a key $k \in \mathbb{Z}_p$ for $\mathsf{2HDH}$. The vector $\boldsymbol{r}$ holds the blinding factors.

```
Party 1                                                          Party 2
Input: $\boldsymbol{x}$                                          Input: $k$

For $j = 1, \ldots, |\boldsymbol{x}|$ do
  $\boldsymbol{r}[j] \leftarrow_\$ \mathbb{Z}_p^*$ ; $\boldsymbol{b}[j] \leftarrow \mathsf{H}(1,x)^{\boldsymbol{r}[j]}$
                                  $\xrightarrow{\boldsymbol{b}}$   For $j = 1, \ldots, |\boldsymbol{b}|$ do
                                  $\xleftarrow{g^k, \boldsymbol{z}}$     $\boldsymbol{z}[j] \leftarrow \boldsymbol{b}[j]^k$
For $j = 1, \ldots, |\boldsymbol{x}|$ do
  $\boldsymbol{u}[j] \leftarrow \boldsymbol{z}[j]^{1/\boldsymbol{r}[j]}$
  $\boldsymbol{y}[j] \leftarrow \mathsf{H}(2, g^k, \boldsymbol{x}[j], \boldsymbol{u}[j])$
Output: $\boldsymbol{y}$                                         Output: $|\boldsymbol{b}|$
```

Fig. 13. 2H-DH OPRF protocol Π^{2HDH}.

InI security for the client. We first want to show InI security of the Π^{2HDH} OPRF protocol for an honest client (party 1). In our concrete setting, we aim to give a concrete bound on the ini-advantage of adversary A_{ini}. The proof of the following is in [12].

Theorem 9. *Let $\mathbb{G} = \langle g \rangle$ be a group with prime order p, and $\ell \geq 1$ an integer. Let $\mathsf{2HDH}$ be the associated 2H-DH family of functions as per Fig. 3. Let Π^{2HDH} be the associated 2H-DH OPRF protocol as per Fig. 13 and let $\mathsf{F} = \mathsf{F}^{\mathrm{oprf}}_{\mathsf{2HDH}}$ be the OPRF functionality that Π^{2HDH} computes. Let A_{ini} be an adversary playing game $\mathbf{G}^{\mathrm{ini}}_{\mathsf{F},\Pi^{\mathsf{2HDH}},1}$. Then*

$$\mathbf{Adv}^{\mathrm{ini}}_{\mathsf{F},\Pi^{\mathsf{2HDH}},1}(A_{\mathrm{ini}}) \leq \frac{\mathrm{Q}^{\mathrm{RO}}(A_{\mathrm{ini}})}{p} \ . \tag{14}$$

OPRF-PR security for the server. We bound the adversary advantage via the advantage to solve the different problems from Sect. 5 on the underlying group $\mathbb{G} = \langle g \rangle$, showcasing how the bounds change across these problems. The proof of the following is in [12].

Theorem 10. *Let $\mathbb{G} = \langle g \rangle$ be a group with prime order p, and $\ell \geq 1$ an integer. Let $\mathsf{2HDH}$ be the associated 2H-DH family of functions as per Fig. 3. Let Π^{2HDH} be the associated 2H-DH OPRF protocol as per Fig. 13. Let A_{oprf} be an adversary* *playing game* $\mathbf{G}^{\text{oprf-pr}}_{\Pi^{2HDH},2HDH}$*, and let* $\text{xx} \in \{\text{ddh}, \text{cdh}, \text{v-cdh}, \text{cdh-muc}, \text{v-cdh-muc}\}$*. Then we can construct an adversary* A_{xx} *playing game* $\mathbf{G}^{\text{xx}}_{\mathbb{G},g,p}$ *such that*

$$\begin{aligned}\mathbf{Adv}^{\text{oprf-pr}}_{\Pi^{2HDH},2HDH}(A_{\text{oprf}}) \le 2 \cdot \boldsymbol{\mu}^{\text{xx}}(q_{\text{ro}}, q_{\text{n}}) \cdot \mathbf{Adv}^{\text{xx}}_{\mathbb{G},g,p}(A_{\text{xx}}) \\ + \frac{2 \cdot \boldsymbol{\delta}^{\text{xx}}(q_{\text{ro}}, q_{\text{n}})}{p}\ ,\end{aligned} \tag{15}$$

where $q_{\text{n}} = \text{Q}^{\text{New}}(A_{\text{oprf}})$ *and* $q_{\text{ro}} = \text{Q}^{\text{RO}}(A_{\text{oprf}})$*. Further*

$$\boldsymbol{\mu}^{\text{xx}}(q_{\text{ro}}, q_{\text{n}}) = \begin{cases} q_{\text{ro}}^2 q_{\text{n}} & \textit{if}\ \text{xx} = \text{cdh} \\ q_{\text{ro}} q_{\text{n}} & \textit{if}\ \text{xx} = \text{v-cdh} \\ q_{\text{ro}} & \textit{if}\ \text{xx} = \text{cdh-muc} \\ 1 & \textit{if}\ \text{xx} \in \{\text{v-cdh-muc}, \text{ddh}\} \end{cases} \tag{16}$$

and

$$\boldsymbol{\delta}^{\text{xx}}(q_{\text{ro}}, q_{\text{n}}) = \begin{cases} q_{\text{ro}} \cdot q_{\text{n}} & \textit{if}\ \text{xx} \in \{\text{cdh}, \text{v-cdh}, \text{cdh-muc}, \text{v-cdh-muc}\} \\ q_{\text{ro}} \cdot q_{\text{n}} + q_{\text{ro}} + 1 & \textit{if}\ \text{xx} = \text{ddh} \end{cases}$$

with resources

$$\begin{aligned} \text{Q}^{\text{NewKey}}(A_{\text{cdh-muc}}) = \text{Q}^{\text{NewKey}}(A_{\text{v-cdh-muc}}) &= \text{Q}^{\text{New}}(A_{\text{oprf}}) \\ \text{Q}^{\text{NewBase}}(A_{\text{cdh-muc}}) = \text{Q}^{\text{NewBase}}(A_{\text{v-cdh-muc}}) &\le \text{Q}^{\text{RO}}(A_{\text{oprf}})\ , \\ \text{Q}^{\text{CDHO}}(A_{\text{cdh-muc}}) = \text{Q}^{\text{CDHO}}(A_{\text{v-cdh-muc}}) &\le \ell_{\text{tr}} \cdot \text{Q}^{\text{TR}}(A_{\text{oprf}}), \\ \text{Q}^{\text{DDHO}}(A_{\text{v-cdh}}) = \text{Q}^{\text{DDHO}}(A_{\text{v-cdh-muc}}) &\le \text{Q}^{\text{RO}}(A_{\text{oprf}}), \end{aligned}$$

where ℓ_{tr} *is the maximum length of vectors queried to the* TR *oracle and the running times of all adversaries are about that of* A_{oprf}*, except that* A_{cdh} *and* $A_{\text{v-cdh}}$ *additionally perform at most* $\text{Q}^{\text{New}}(A_{\text{oprf}}) + \text{Q}^{\text{RO}}(A_{\text{oprf}}) + \ell_{\text{tr}} \cdot \text{Q}^{\text{TR}}(A_{\text{oprf}})$ *group exponentiations and* A_{ddh} *additionally performs at most* $\text{Q}^{\text{New}}(A_{\text{oprf}}) + 2 \cdot \text{Q}^{\text{RO}}(A_{\text{oprf}}) + 2 \cdot \ell_{\text{tr}} \cdot \text{Q}^{\text{TR}}(A_{\text{oprf}})$ *group exponentiations.*

With these results on the security of Π^{2HDH} and Theorem 6, we can show concrete bounds on the InI security of the DH-PSI protocol. (By the latter we mean the PSI protocol in Fig. 10 when using Π^{2HDH} as the underlying OPRF.) An approximate summary of these bounds was given in Fig. 2.

7 The Salted-DH PSI Protocol

We give a new PSI protocol. It has a proof with a *tight* reduction to the V-CDH (and hence also DDH) assumption and achieves better bounds for the CDH assumption than the previous protocol. Yet it has essentially the same computational cost as the OPRF-PSI when the OPRF is 2H-DH. The communication cost is more by just a constant (256 bits in practice) that does not depend on the sizes of the sets in the protocol.

Salted-DH PSI Protocol $\Pi^{\text{salt-psi}}$

Parameters: Salt length sl, hash length hl, group $\mathbb{G}$ of prime order p with generator g
Party 1 input: Set $S_1 \subseteq \{0,1\}^*$ of size s_1
Party 2 input: Set $S_2 \subseteq \{0,1\}^*$ of size s_2
Oracles: $\mathsf{H}_1 : \{0,1\}^{sl} \times \mathbb{G} \to \mathbb{G}$ and $\mathsf{H}_2 : \{0,1\}^{sl} \times \mathbb{G} \times \{0,1\}^* \times \mathbb{G} \to \{0,1\}^{hl}$

1. Party 1 casts the items in its set S_1 as a vector $\boldsymbol{x}_1 \leftarrow \mathsf{S2V}(S_1)$ of length s_1. It picks a salt $\eta \leftarrow\$ \{0,1\}^{sl}$ and picks blinding exponents $r_1, \ldots, r_{s_1} \leftarrow\$ \mathbb{Z}_p^*$. It then sets $\boldsymbol{a}_1[i] \leftarrow \mathsf{H}_1(\eta, \boldsymbol{x}_1[i])^{r_i}$ for $i = 1, \ldots, s_1$. It sends $(\eta, \boldsymbol{a}_1)$ to Party 2.
2. Party 2 picks a key $k \leftarrow\$ \mathbb{Z}_p$ and computes $K \leftarrow g^k$. It lets $\boldsymbol{b}[i] \leftarrow \boldsymbol{a}_1[i]^k$ for $i = 1, \ldots, |\boldsymbol{a}_1|$. It randomly permutes the items in its set S_2 to get a vector $\boldsymbol{x}_2 \leftarrow\$ \mathsf{S2V}(S_2)$ of length s_2. It lets $\boldsymbol{c}[i] \leftarrow \mathsf{H}_1(\eta, \boldsymbol{x}_2[i])^k$ and $\boldsymbol{a}_2[i] \leftarrow \mathsf{H}_2(\eta, K, \boldsymbol{x}_2[i], \boldsymbol{c}[i])$ for $i = 1, \ldots, s_2$. It sends $(K, \boldsymbol{b}, \boldsymbol{a}_2)$ to Party 1. It then halts with $s_1 = |\boldsymbol{a}_1|$ as its protocol output.
3. Party 1 constructs intersection set I via

```
For all $i = 1, \ldots, s_1$ do
    $t_i \leftarrow r_i^{-1} \bmod p$ ; $\boldsymbol{d}[i] \leftarrow \boldsymbol{b}[i]^{t_i}$   // $\boldsymbol{d}[i] = \mathsf{H}_1(\eta, \boldsymbol{x}_1[i])^k$
    If $\mathsf{H}_2(\eta, K, \boldsymbol{x}_1[i], \boldsymbol{d}[i]) \in \mathsf{V2S}(\boldsymbol{a}_2)$ then $I \leftarrow I \cup \{\boldsymbol{x}_1[i]\}$.
```

It outputs I as the intersection of S_1 and S_2, and halts.

Fig. 14. Salted DH PSI protocol $\Pi^{\text{salt-psi}}$.

The protocol is presented in Fig. 14. We have fixed a group $\mathbb{G}$ of prime order p. The protocol is parameterized by a salt length sl and a hash-output length hl. (The latter only impacts correctness, not security.) Compared to 2H-DH based PSI, the increase in computational is just that the parties need to hash slightly longer strings, which has negligible cost relative to the cost of the group exponentiations, which is the same in both protocols. Communication increases by just sl, irrespective of the sizes of the sets involved.

The following Theorem establishes correctness and security of the protocol. The main claim is the third, showing security for the server based only on the V-CDH assumption. The added term is easily made negligible by picking a non-trivial salt length; in practice, $sl = 256$ will do. The result is that the reduction is tight. The proof is in [12].

Theorem 11. *Let $\mathbb{G}$ be a group of prime order p with generator g. Let $hl, sl \geq 0$ be integers. Let $\Pi = \Pi^{\text{salt-psi}}$ be the associated PSI protocol as per Fig. 14. Let F be the PSI functionality over universe $\{0,1\}^*$. Below, M denotes an upper bound on the sum, across all* Run *queries of adversaries A_{corr} and A_{ini}, of the sizes of the sets in these queries.*

1. *Correctness: Let A_{corr} be an adversary playing game $\mathbf{G}^{\text{corr}}_{\mathsf{F},\Pi}$. Then*

$$\mathbf{Adv}^{\text{corr}}_{\mathsf{F},\Pi}(A_{\text{corr}}) \leq M^2/2^{hl} \ . \tag{17}$$

2. *Security for the client: Let A_{ini} be an adversarial server, meaning an adversary playing game $\mathbf{G}^{\text{ini}}_{\mathsf{F},\Pi,1}$. Then*

$$\mathbf{Adv}^{\text{ini}}_{\mathsf{F},\Pi,1}(A_{\text{ini}}) \leq \frac{\mathrm{Q}^{\text{RO}}(A_{\text{ini}})}{p} \ . \tag{18}$$

3. *Security for the server: Let A_{ini} be an adversarial client, meaning an adversary playing game $\mathbf{G}^{\mathrm{ini}}_{\mathsf{F},\Pi,2}$. Then we can construct an adversary A_{xx} playing game $\mathbf{G}^{\mathrm{xx}}_{\mathbb{G},g,p}$ such that*

$$\begin{aligned}\mathbf{Adv}^{\mathrm{ini}}_{\mathsf{F},\Pi,2}(A_{\mathrm{ini}}) \le 2 \cdot \boldsymbol{\mu}^{\mathrm{xx}}(q_{\mathrm{ro}}) \cdot \mathbf{Adv}^{\mathrm{xx}}_{\mathbb{G},g,p}(A_{\mathrm{xx}}) \\ + \frac{2 \cdot q_{\mathrm{rn}}(q_{\mathrm{rn}} + q_{\mathrm{ro}})}{2^{sl}} + \frac{2 \cdot \boldsymbol{\delta}^{\mathrm{xx}}(q_{\mathrm{ro}})}{p} \ .\end{aligned} \tag{19}$$

where $q_{\mathrm{rn}} = \mathrm{Q}^{\mathrm{RUN}}(A_{\mathrm{ini}})$ and $q_{\mathrm{ro}} = \mathrm{Q}^{\mathrm{RO}}(A_{\mathrm{ini}})$. Further

$$\boldsymbol{\mu}^{\mathrm{xx}}(q_{\mathrm{ro}}) = \begin{cases} q_{\mathrm{ro}} & \textit{if}\ \mathrm{xx} \in \{\mathrm{cdh}, \mathrm{cdh\text{-}muc}\} \\ 1 & \textit{if}\ \mathrm{xx} \in \{\mathrm{v\text{-}cdh}, \mathrm{v\text{-}cdh\text{-}muc}, \mathrm{ddh}\} \end{cases} \tag{20}$$

and

$$\boldsymbol{\delta}^{\mathrm{xx}}(q_{\mathrm{ro}}) = \begin{cases} 0 & \textit{if}\ \mathrm{xx} \in \{\mathrm{cdh}, \mathrm{v\text{-}cdh}, \mathrm{cdh\text{-}muc}, \mathrm{v\text{-}cdh\text{-}muc}\} \\ q_{\mathrm{ro}} + 1 & \textit{if}\ \mathrm{xx} = \mathrm{ddh} \end{cases}$$

with resources

$$\begin{aligned} \mathrm{Q}^{\mathrm{NEWKEY}}(A_{\mathrm{cdh\text{-}muc}}) = \mathrm{Q}^{\mathrm{NEWKEY}}(A_{\mathrm{v\text{-}cdh\text{-}muc}}) &= 1 \\ \mathrm{Q}^{\mathrm{NEWBASE}}(A_{\mathrm{cdh\text{-}muc}}) = \mathrm{Q}^{\mathrm{NEWBASE}}(A_{\mathrm{v\text{-}cdh\text{-}muc}}) &= 1\ , \\ \mathrm{Q}^{\mathrm{CDHO}}(A_{\mathrm{cdh\text{-}muc}}) = \mathrm{Q}^{\mathrm{CDHO}}(A_{\mathrm{v\text{-}cdh\text{-}muc}}) &= 0\ , \\ \mathrm{Q}^{\mathrm{DDHO}}(A_{\mathrm{v\text{-}cdh}}) = \mathrm{Q}^{\mathrm{DDHO}}(A_{\mathrm{v\text{-}cdh\text{-}muc}}) &\le \mathrm{Q}^{\mathrm{RO}}(A_{\mathrm{ini}})\ . \end{aligned}$$

The running times of $A_{\mathrm{v\text{-}cdh\text{-}muc}}$ and $A_{\mathrm{cdh\text{-}muc}}$ are about that of A_{ini}. The adversaries $A_{\mathrm{v\text{-}cdh}}$ and $A_{\mathrm{v\text{-}cdh}}$ perform an additional $\mathrm{Q}^{\mathrm{RUN}}(A_{\mathrm{ini}}) + M + \mathrm{Q}^{\mathrm{RO}}(A_{\mathrm{ini}})$ group exponentiations and A_{ddh} performs an additional $\mathrm{Q}^{\mathrm{RUN}}(A_{\mathrm{ini}}) + 2 \cdot M + 2 \cdot \mathrm{Q}^{\mathrm{RO}}(A_{\mathrm{ini}})$ group exponentiations.

Acknowledgments. Bellare and Ranjan were supported in part by NSF grant CNS-2154272 and KACST. Work done while Riepel was at UCSD, supported in part by KACST.

References

1. Abdalla, M., Bellare, M., Rogaway, P.: The oracle Diffie-Hellman assumptions and an analysis of DHIES. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 143–158. Springer, Heidelberg (Apr 2001)
2. Agrawal, S., Agrawal, S., Prabhakaran, M.: Cryptographic agents: Towards a unified theory of computing on encrypted data. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 501–531. Springer, Heidelberg (Apr 2015)
3. Apple: Password monitoring. https://support.apple.com/guide/security/password-monitoring-sec78e79fc3b/web (Feb 2021)

4. Baldi, P., Baronio, R., De Cristofaro, E., Gasti, P., Tsudik, G.: Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) ACM CCS 2011. pp. 691–702. ACM Press (Oct 2011)
5. Bellare, M.: A concrete-security analysis of the apple psi protocol. https://www.apple.com/child-safety/pdf/Alternative_Security_Proof_of_Apple_PSI_System_Mihir_Bellare.pdf (July 2021)
6. Bellare, M., Canetti, R., Krawczyk, H.: Pseudorandom functions revisited: The cascade construction and its concrete security. In: 37th FOCS. pp. 514–523. IEEE Computer Society Press (Oct 1996)
7. Bellare, M., Hoang, V.T., Keelveedhi, S., Rogaway, P.: Efficient garbling from a fixed-key blockcipher. In: 2013 IEEE Symposium on Security and Privacy. pp. 478–492. IEEE Computer Society Press (May 2013)
8. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 2012. pp. 784–796. ACM Press (Oct 2012)
9. Bellare, M., Kilian, J., Rogaway, P.: The security of cipher block chaining. In: Desmedt, Y. (ed.) CRYPTO'94. LNCS, vol. 839, pp. 341–358. Springer, Heidelberg (Aug 1994)
10. Bellare, M., Namprempre, C., Pointcheval, D., Semanko, M.: The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. Journal of Cryptology 16(3), 185–215 (Jun 2003)
11. Bellare, M., O'Neill, A.: Semantically-secure functional encryption: Possibility results, impossibility results and the quest for a general definition. In: Abdalla, M., Nita-Rotaru, C., Dahab, R. (eds.) CANS 13. LNCS, vol. 8257, pp. 218–234. Springer, Heidelberg (Nov 2013)
12. Bellare, M., Ranjan, R., Riepel, D., Aldakheel, A.: The concrete security of two-party computation: Simpler definitions, and tight proofs for psi and oprfs. Cryptology ePrint Archive (2024), full version of this paper
13. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Denning, D.E., Pyle, R., Ganesan, R., Sandhu, R.S., Ashby, V. (eds.) ACM CCS 93. pp. 62–73. ACM Press (Nov 1993)
14. Bellare, M., Rogaway, P.: The exact security of digital signatures: How to sign with RSA and Rabin. In: Maurer, U.M. (ed.) EUROCRYPT'96. LNCS, vol. 1070, pp. 399–416. Springer, Heidelberg (May 1996)
15. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 409–426. Springer, Heidelberg (May / Jun 2006)
16. Blum, M., Micali, S.: How to generate cryptographically strong sequences of pseudorandom bits. SIAM Journal on Computing 13(4), 850–864 (1984)
17. Boldyreva, A.: Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In: Desmedt, Y. (ed.) PKC 2003. LNCS, vol. 2567, pp. 31–46. Springer, Heidelberg (Jan 2003)
18. Boneh, D., Sahai, A., Waters, B.: Functional encryption: Definitions and challenges. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 253–273. Springer, Heidelberg (Mar 2011)
19. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press (Oct 2001)
20. Chase, M., Miao, P.: Private set intersection in the internet setting from lightweight oblivious PRF. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part III. LNCS, vol. 12172, pp. 34–63. Springer, Heidelberg (Aug 2020)

21. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 1243–1255. ACM Press (Oct / Nov 2017)
22. Davidson, A., Goldberg, I., Sullivan, N., Tankersley, G., Valsorda, F.: Privacy pass: Bypassing internet challenges anonymously. PoPETs 2018(3), 164–180 (Jul 2018)
23. Everspaugh, A., Chatterjee, R., Scott, S., Juels, A., Ristenpart, T.: The pythia PRF service. In: Jung, J., Holz, T. (eds.) USENIX Security 2015. pp. 547–562. USENIX Association (Aug 2015)
24. Feige, U., Lapidot, D., Shamir, A.: Multiple non-interactive zero knowledge proofs based on a single random string (extended abstract). In: 31st FOCS. pp. 308–317. IEEE Computer Society Press (Oct 1990)
25. Freedman, M.J., Ishai, Y., Pinkas, B., Reingold, O.: Keyword search and oblivious pseudorandom functions. In: Kilian, J. (ed.) TCC 2005. LNCS, vol. 3378, pp. 303–324. Springer, Heidelberg (Feb 2005)
26. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Cachin, C., Camenisch, J. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 1–19. Springer, Heidelberg (May 2004)
27. Goldreich, O.: Foundations of Cryptography: Basic Tools, vol. 1. Cambridge University Press, Cambridge, UK (2001)
28. Goldreich, O.: Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press, Cambridge, UK (2004)
29. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. Journal of the ACM 33(4), 792–807 (Oct 1986)
30. Goldwasser, S., Micali, S.: Probabilistic encryption. Journal of Computer and System Sciences 28(2), 270–299 (1984)
31. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM Journal on Computing 18(1), 186–208 (1989)
32. Hazay, C., Lindell, Y.: Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In: Canetti, R. (ed.) TCC 2008. LNCS, vol. 4948, pp. 155–175. Springer, Heidelberg (Mar 2008)
33. Hunt, T., Hunt, C., Siguroarson, S.: Have I been pwned? https://haveibeenpwned.com/
34. Ion, M., Kreuter, B., Nergiz, A.E., Patel, S., Raykova, M., Saxena, S., Seth, K., Shanahan, D., Yung, M.: On deploying secure computing commercially: Private intersection-sum protocols and their business applications. Cryptology ePrint Archive, Report 2019/723 (2019), https://eprint.iacr.org/2019/723
35. Ion, M., Kreuter, B., Nergiz, E., Patel, S., Saxena, S., Seth, K., Shanahan, D., Yung, M.: Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738 (2017), https://eprint.iacr.org/2017/738
36. Jarecki, S., Kiayias, A., Krawczyk, H.: Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part II. LNCS, vol. 8874, pp. 233–253. Springer, Heidelberg (Dec 2014)
37. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 456–486. Springer, Heidelberg (Apr / May 2018)
38. Jarecki, S., Liu, X.: Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 577–594. Springer, Heidelberg (Mar 2009)

39. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: A low latency framework for secure neural network inference. In: Enck, W., Felt, A.P. (eds.) USENIX Security 2018. pp. 1651–1669. USENIX Association (Aug 2018)
40. Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious PRF with applications to private set intersection. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 818–829. ACM Press (Oct 2016)
41. Kolesnikov, V., Matania, N., Pinkas, B., Rosulek, M., Trieu, N.: Practical multi-party private set intersection from symmetric-key techniques. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 1257–1272. ACM Press (Oct / Nov 2017)
42. Li, L., Pal, B., Ali, J., Sullivan, N., Chatterjee, R., Ristenpart, T.: Protocols for checking compromised credentials. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 1387–1403. ACM Press (Nov 2019)
43. Lindell, Y.: How to simulate it - A tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046 (2016), https://eprint.iacr.org/2016/046
44. Marlinspike, M.: The difficulty of private contact discovery. https://signal.org/blog/contact-discovery/ (Jan 2014)
45. Mezzour, G., Perrig, A., Gligor, V.D., Papadimitratos, P.: Privacy-preserving relationship path discovery in social networks. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 09. LNCS, vol. 5888, pp. 189–208. Springer, Heidelberg (Dec 2009)
46. Naor, M., Reingold, O.: Number-theoretic constructions of efficient pseudo-random functions. In: 38th FOCS. pp. 458–467. IEEE Computer Society Press (Oct 1997)
47. Narayanan, A., Thiagarajan, N., Lakhani, M., Hamburg, M., Boneh, D.: Location privacy via private proximity testing. In: NDSS 2011. The Internet Society (Feb 2011)
48. Okamoto, T., Pointcheval, D.: The gap-problems: A new class of problems for the security of cryptographic schemes. In: Kim, K. (ed.) PKC 2001. LNCS, vol. 1992, pp. 104–118. Springer, Heidelberg (Feb 2001)
49. O'Neill, A.: Definitional issues in functional encryption. Cryptology ePrint Archive, Report 2010/556 (2010), https://eprint.iacr.org/2010/556
50. Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: SpOT-light: Lightweight private set intersection from sparse OT extension. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 401–431. Springer, Heidelberg (Aug 2019)
51. Pinkas, B., Schneider, T., Segev, G., Zohner, M.: Phasing: Private set intersection using permutation-based hashing. In: Jung, J., Holz, T. (eds.) USENIX Security 2015. pp. 515–530. USENIX Association (Aug 2015)
52. Rabin, M.O.: How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187 (2005), https://eprint.iacr.org/2005/187
53. Rajan, A., Qin, L., Archer, D.W., Boneh, D., Lepoint, T., Varia, M.: Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In: Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies. pp. 1–4 (2018)
54. Thomas, K., Pullman, J., Yeo, K., Raghunathan, A., Kelley, P.G., Invernizzi, L., Benko, B., Pietraszek, T., Patel, S., Boneh, D., Bursztein, E.: Protecting accounts from credential stuffing with password breach alerting. In: Heninger, N., Traynor, P. (eds.) USENIX Security 2019. pp. 1556–1571. USENIX Association (Aug 2019)

55. Tyagi, N., Celi, S., Ristenpart, T., Sullivan, N., Tessaro, S., Wood, C.A.: A fast and simple partially oblivious PRF, with applications. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II. LNCS, vol. 13276, pp. 674–705. Springer, Heidelberg (May / Jun 2022)
56. Yao, A.C.C.: Protocols for secure computations (extended abstract). In: 23rd FOCS. pp. 160–164. IEEE Computer Society Press (Nov 1982)

Blockchain Protocols

Jackpot: Non-interactive Aggregatable Lotteries

Nils Fleischhacker[1(✉)], Mathias Hall-Andersen[2], Mark Simkin[3], and Benedikt Wagner[4]

[1] Ruhr University Bochum, Bochum, Germany
mail@nilsfleischhacker.de
[2] ZkSecurity, New York, USA
mathias@zksecurity.xyz
[3] Berlin, Germany
msimkin@gmx.de
[4] Ethereum Foundation, Berlin, Germany
benedikt.wagner@ethereum.org

Abstract. In proof-of-stake blockchains, liveness is ensured by repeatedly selecting random groups of parties as leaders, who are then in charge of proposing new blocks and driving consensus forward. The lotteries that elect those leaders need to ensure that adversarial parties are not elected disproportionately often and that an adversary can not tell who was elected before those parties decide to speak, as this would potentially allow for denial-of-service attacks. Whenever an elected party speaks, it needs to provide a winning lottery ticket, which proves that the party did indeed win the lottery. Current solutions require all published winning tickets to be stored individually on-chain, which introduces undesirable storage overheads.

In this work, we introduce *non-interactive aggregatable lotteries* and show how these can be constructed efficiently. Our lotteries provide the same security guarantees as previous lottery constructions, but additionally allow any third party to take a set of published winning tickets and aggregate them into one short digest. We provide a formal model of our new primitive in the universal composability framework.

As one of our technical contributions, which may be of independent interest, we introduce aggregatable vector commitments with simulation-extractability and present a concretely efficient construction thereof in the algebraic group model in the presence of a random oracle. We show how these commitments can be used to construct non-interactive aggregatable lotteries. We have implemented our construction, called *Jackpot*, and provide benchmarks that underline its concrete efficiency.

M. Hall-Andersen—This work was done while this author was at Aarhus University. Funded by the Concordium Foundation.
B. Wagner—This work was done while the author was at CISPA Helmholtz Center for Information Security. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 507237585.
M. Simkin—Independent Researcher.

Keywords: Lotteries · Aggregation · Vector Commitments · Simulation-Extractability · KZG Commitments

1 Introduction

Blockchains rely on lottery mechanisms for repeatedly electing one or multiple leaders at random from the pool of all participants. These leaders are then in charge of proposing new blocks and driving the protocol's consensus forward, thereby ensuring liveness of the blockchain. In proof-of-stake blockchains, the participants' probabilities of being elected are tied to their stake, i.e., to the amount of money they have put into the system. In Ethereum, each participant deposits a fixed amount of money to participate in the lotteries and thus everybody has the same probability of being elected. In Algorand [26], on the other hand, participants may have deposited different amounts of money and therefore have different probabilities of being elected.

In the context of proof-of-stake blockchains a lottery mechanism needs to satisfy several properties. From a security perspective, lotteries should not allow corrupt parties to be elected disproportionately often. Lotteries should hide who the elected leaders are, as an adversary could otherwise prevent the chain from growing by taking the leaders off the network right after they have been elected, but before they have had a chance to speak. Leaders should privately learn whether they won the lottery and obtain a publicly verifiable winning ticket. When a leader is ready to speak, they can attach the winning ticket to their message, so that everybody can verify that they are indeed one of the leaders.

From an efficiency perspective, lotteries should aim to minimize both the network bandwidth and storage overheads that they incur, since new leaders may need to be elected frequently among a large number of participants. In terms of bandwidth overhead we would like to minimize the amount of communication needed to run each lottery. In terms of storage overhead we would like to minimize the amount of memory needed to store all published winning tickets. Ideally, we would like the storage overhead to grow sublinearly in the number of published winning tickets.

Various constructions of lotteries schemes have already been proposed in the literature, but all of them either do not keep the lottery output secret [1,4,11], require a trusted party [16,31], or have storage overheads that are linear in the number of published winning tickets [17,26] per election.

1.1 Our Contribution

In this work, we introduce *non-interactive aggregatable lotteries*. In this setting we have a set of parties where each party is identified by a short verification key and holds a corresponding secret key. We assume the existence of a randomness beacon functionality which broadcasts uniformly random values to all parties in regular intervals. We will associate the randomness beacon output at time t with the t-th lottery execution.

Whenever the randomness beacon outputs a lottery seed, every party can, without interacting with the other parties, check whether they have won the current lottery. Each party will win each lottery independently with probability $1/k$ for some fixed parameter[1] k. Maliciously generated keys do not allow the adversary to increase their winning probabilities or to coordinate which corrupt parties win which lotteries at which times. The adversary is not able to determine which honest parties are winning which elections with probability noticeably better than guessing. Each winning party can locally compute a publicly verifiable proof, the winning ticket, that allows them to convince other parties that they won a lottery. Finally, and most importantly, the lotteries are aggregatable. By this we mean that all published winning tickets belonging to the same lottery execution can be compressed into one short ticket by any (possibly untrusted) third party. Given the public keys of all winning parties and the compressed lottery ticket anybody can still be convinced of the fact that each individual party won the lottery. We formally model these lotteries in the universal composability (UC) framework of Canetti [13].

Lotteries from Simulation-Extractable Vector Commitments. We introduce the notion of aggregatable vector commitments with a strong simulation-extractability property and show that these commitments can be used to instantiate our non-interactive aggregate lotteries. On an intuitive level, a vector commitment is said to be aggregatable if openings belonging to different commitments can be compressed into one short opening. A vector commitment is said to be simulation-extractable if it satisfies the following two properties: in security proofs, knowing a trapdoor, we can issue dummy commitments and later open those to arbitrary messages at arbitrary positions. Additionally we can extract the committed messages from any valid but adversarially chosen commitment. While our notion effectively requires the commitments to be "non-malleable", the openings of such a commitment scheme can still have homomorphic properties, which is of crucial importance for being able to aggregate them.

Simulation-Extractable Vector Commitments from KZG. We present a construction of such an aggregatable vector commitment with simulation-extractability proven secure in the algebraic group model (AGM) [24]. Our construction is a modification of the polynomial commitment scheme of Kate, Zaverucha, and Goldberg (KZG) [30] and uses the exact same trusted setup. While KZG itself is malleable and can therefore not be simulation-extractable, we show that our construction is simulation-extractable. At the same time it preserves the homomorphic properties of KZG needed for aggregation. The proof turns out to be rather involved and we present it in a modular way. We believe that our construction, our notion of simulation-extractability, and our modular proof may be of independent interest beyond their applications in this work.

Implementation and Benchmarks. To show the practicality of our construction, called *Jackpot*, we have implemented it and provide benchmarks for various parameter settings. For instance, Jackpot allows for aggregating 2048 winning tickets in less than 15 milliseconds and verifying the aggregated ticket takes less than 17 milliseconds on a regular Macbook Pro. Storing the 2048 winning tickets

[1] We also show how to generalize our notion of lotteries and our constructions to the setting where parties have different winning probabilities.

in aggregated form is 1228.8 times more efficient than storing a list of all tickets of a state-of-the-art lottery based on VRFs explicitly. The main bottleneck of our construction is the time it takes to generate the public keys. For generating a public key that is good for 2^{20} lotteries, i.e., for one lottery every 5 minutes for 10 years nonstop, our protocol takes around 8 seconds. The corresponding public key is 160 bytes large.

1.2 Related Work

Lotteries have appeared throughout cryptographic research in various shapes and forms. In the following we discuss a few of those research works and highlight how they differ from ours.

Lotteries without Secrecy. The problem of allowing a group of parties to select a random set of leaders among them has already been addressed by Broder and Dolev [11] over 40 years ago. Their work, however, requires a large amount of interaction during each election and does not hide who is elected. The works of Bentov and Kumaresan [4] and of Bartoletti and Zunino [1] allow parties to run financial lotteries that enjoy certain fairness properties on top of cryptocurrencies like Bitcoin or Ethereum. Here each party can deposit a coin and a random parties is elected to be the winner that obtains all deposited coins. Neither of those protocols provides any privacy guarantees and their techniques do not seem applicable to our setting.

Lotteries without Aggregation. A lottery that satisfies all of our desired properties apart from aggregation was proposed by Gilad et al. [26]. In their construction each party is identified via a public key for a verifiable random functions (VRF) [37]. The public key of party i can be viewed as a commitment to a secret random function f_i and, using their corresponding secret key, party i is able to output pairs (x, y) and prove that $y = f_i(x)$. Whenever a randomness beacon provides lseed, party i can check whether they won the corresponding lottery by computing whether $f_i(\mathsf{lseed}) < k$ for some parameter k. Since the function is random, nobody can predict whether party i wins a lottery. At the same time the verifiability property of the random function allows party i to claim the win. In subsequent work David et al. [17] properly formalized this approach and showed that the VRF actually needs to satisfy an additional property ensuring that high entropy inputs produce high entropy outputs even if the VRF keys were chosen by a malicious party.

Both works [17,26] show different ways of how their lotteries can then be used to select committees that then drive consensus forward in their respective blockchain designs. Both works would benefit from being able to aggregate lottery tickets as it would allow them to reduce their storage complexities.

Single Secret Leader Elections. A recent work by Boneh et al. [7] introduces the problem of secret leader elections and shows how it can be solved using cryptographic tools like indistinguishability obfuscation [25], threshold fully homomorphic encryption [8], or proofs of correct shuffles [2]. Whereas our work focuses on electing a certain number of leaders *in expectation*, they focus on computing an ordered list of an *exact* number of leaders. As their problem is significantly harder to solve, their protocols are significantly more expensive computationally and require large amounts of interaction for each lottery.

Aggregatable Vector Commitments. We mentioned above that our main technical tool is an aggregatable vector commitment that satisfies a strong form of simulation-extractability. Various aggregatable or linearly homomorphic vector commitments [14,22,23,27,32–34,38,41] have previously been proposed but all of these works fail to achieve simulation-extractability which is of crucial importance for our application.

On a technical level a recent result by Faonio et al. [18] uses some observations similar to ours. They construct simulation-extractable succinct non-interactive arguments of knowledge (SNARKs) [28,39]. To this end they show that the KZG polynomial commitment scheme satisfies a weak notion of simulation-extractability in the AGM. Indeed, there is no hope of proving full simulation-extractability for KZG commitments as both commitments and openings are homomorphic. Conceptually, both our work and theirs show that opening KZG at a random point chosen after the commitment is fixed makes the commitment simulation-extractable. However, we highlight three important differences: firstly, the notion that they show for the original KZG construction is tailored to their specific use-case in SNARKs. Contrary to that, we define a new scheme and show a full simulation-extractability notion that is more self-contained. Secondly, their notion states that we can extract a preimage from a KZG commitment (under certain restrictions), whereas our notion additionally guarantees that any future (aggregated) opening provided by the adversary is consistent with the extracted preimage. We can view this as a new form of binding for aggregated openings that composes with extraction. Interestingly, Faonio et al. also need a binding property, but only implicitly show it during the compilation to a SNARK. Finally, our analysis is more modular: we manage to generically separate simulation, extractability, and binding aspects.

In a concurrent and independent work, Libert [35] also constructs a simulation-extractable version of KZG commitments. However, the goals of our work and Libert's work are orthogonal: Libert's construction allows to commit to multivariate polynomials and can be used in HyperPlonk [15]. At the same time, openings can not be aggregated, which is an essential feature of our construction. Indeed, openings in Libert's construction contain a non-interactive Schnorr-style proof [40]. While such proofs can be batched while they are created, it is not possible to aggregate given proofs publicly.

1.3 Technical Overview

One way of instantiating VRFs for lotteries that rely on them, e.g. [17,26], is to use the unique signature scheme of Boneh, Lynn, and Shacham (BLS) [10] as a verifiable unpredictable function and then apply a random oracle to the signature to make the output pseudorandom. More concretely, whenever the randomness beacon outputs the unpredictable lottery seed lseed, each participant j signs

lseed (as well as potentially additional context such as their own identity) using their BLS signing key sk_j resulting in a unique signature σ_j. Participant j wins the lottery iff $\mathsf{H}(\sigma_j) < t$, where H is a random oracle and t is an appropriate threshold to achieve the desired winning probability. To prove that they won, the party presents σ_j as their winning ticket. Anyone can verify that they won by verifying the signature using the BLS public key pk_j and checking that indeed $\mathsf{H}(\sigma_j) < t$.

When considering the possibility of aggregating winning tickets, the use of BLS might seem promising at first glance. After all, BLS signatures are known to be aggregatable [9] even in the presence of rogue keys [6] by computing a random linear combination of the signatures. One might thus be tempted to store this short aggregated signature σ instead of a long list of all individual signatures. Alas, this does not work. Although we could still verify that all aggregated σ_j were valid, the exact values of the individual signatures would be lost. We therefore could not recompute their individual hash values to check that all aggregated tickets were winning tickets.

The first idea to solve this dilemma is to try to avoid using the random oracle and directly look for a VRF with nice linearity properties. Specifically, let $(\mathsf{pk}_j, \mathsf{sk}_j)$ be key pairs of a VRF and let $y_j = \mathsf{VRF}(\mathsf{sk}_j, x)$. Further, let τ_j be proofs of the former equality. Then, we want that the following holds for arbitrary weights ξ_j:

$$\mathsf{VRF.Ver}\Big(\sum_{j=1}^{n} \xi_j \mathsf{pk}_j, x, \sum_{j=1}^{n} \xi_j y_j, \sum_{j=1}^{n} \xi_j \tau_j\Big) = 1 \tag{1}$$

The ith round of the lottery could now proceed as follows: given lseed, derive per party challenges x_j. Party j wins the lottery iff $\mathsf{VRF}(\mathsf{sk}_j, (i, \mathsf{lseed})) = x_j$. The corresponding winning ticket is the proof τ_j. Using the linearity of the VRF, we could aggregate the proofs by computing a random linear combination of the winning tickets and weights $(\xi_1, \ldots, \xi_n)$, which are obtained by hashing the set of public keys. The aggregated ticket $\tau = \sum_{i=1}^{n} \xi_j \tau_j$ allows full verification of all proofs via Eq. (1) simultaneously.

For this construction to be sensible we would, however, require a linearly homomorphic VRF with small codomain. Specifically, to achieve a winning probability of $1/k$, the VRF needs a codomain of size exactly k. There are currently no known constructions of such VRFs for usefully small values of k. Fortunately, we can still make the above approach work, if we are willing to make some concessions, namely that a public key will only be valid for a limited number T of successive lotteries. Since T can be chosen sufficiently large for practical purposes and because we can simply generate fresh keys after T lotteries, the concession we make is rather small.

Naive Homomorphic VRFs via Vector Commitments. If we use a vector commitment to commit to a uniformly random vector $\mathbf{v} \in [k]^T$, it can in many ways be viewed as a VRF with domain $[T]$ and codomain $[k]$. The public key is now the commitment and the secret key is the vector $\mathbf{v}$ as well as the randomness used to commit. To participate in T lotteries each party j initially commits to a random vector $\mathbf{v}^{(j)} \in [k]^T$. In the ith lottery round we again derive per party challenges x_j from lseed and party j wins iff $\mathbf{v}_i^{(j)} = x_j$. Each party can *prove* that they won by revealing an opening for position i of their commitment. If the vector commitment has the required homomorphic properties of Eq. (1), we can verify all openings using only the aggregated opening. Luckily for us, such linearly homomorphic vector commitments do exist, with KZG [30] being the most prominent among them.

The Woes of Universal Composability. For our lottery scheme to be useful as part of more complex protocols, it is necessary that it composes securely with itself and other protocols. To this end, we define the security of a lottery scheme in the universal composability (UC) framework [13]. This, however, causes issues with the proof of the construction sketched above. Namely, in the security proof the simulator would need to both equivocate commitments for honest participants and extract from commitments of corrupted participants. This implies that the vector commitment requires some kind of simulation-extractability, i.e., a guarantee that it is possible to extract preimages from any valid commitment produced by an adversary, even if the adversary was previously given equivocal commitments (from which extraction would not be possible).

Unfortunately, not only does KZG not have this property, the required simulation-extractability and the linear homomorphism described above in fact contradict each other. Let com be a valid *simulated* commitment and let τ be an opening proving that com contained x at position i. Then by the linear homomorphism $\mathsf{com}' = \mathsf{com} + \mathsf{com}$ is also a valid commitment and $\tau' = \tau + \tau$ could be used to prove that com' contained $x + x$ at position i. However, it would not be possible to extract a preimage from com'. We thus need to depart from using a regular linear homomorphism for aggregation.

Making KZG Simulation-Extractable. To get around this problem, we make the commitments non-malleable, while maintaining the linear homomorphism on the openings (and a part of the commitments). An expensive black-box way of achieving this might be to add a simulation-extractable proof of knowledge of the secret vector to the commitment. Instead, we can leverage the fact that KZG is not just a vector commitment, but a polynomial commitment. When KZG is used to commit to a vector $\mathbf{v} \in [k]^T$, we are actually committing to the polynomial f of degree $T-1$ over a large field $\mathbb{F}$ that is uniquely defined by the points $(j, \mathbf{v}_j)$. While we have only explicitly defined f on $[T] \subset \mathbb{F}$, we can still open the commitment at any position in $\mathbb{F}$. Now, the idea is to force anyone presenting a fresh commitment to also open their commitment at a random position. If the commitment is derived from simulated commitments, then providing such an opening should not be possible. Since this is an additional opening we need to increase the degree of the polynomial to T and they will turn out that a technicality in the proof actually requires the degree to be $T+1$. The actual construction of our simulation-extractable vector commitment will work as follows: to commit to a vector $\mathbf{v} \in \mathbb{F}^T$ we uniformly choose a polynomial f of degree $T+1$ conditioned on $f(j) = \mathbf{v}_j$ for $j \in [T]$ and commit to it using a regular

KZG commitment $\mathsf{com}_{\mathsf{KZG}}$. The full commitment then consists of $\mathsf{com}_{\mathsf{KZG}}$ as well as an opening of the commitment at position $\mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$ where H is a random oracle mapping to $\mathbb{F}$. The idea is that whenever an adversary would derive a commitment from existing commitments, they would need to open their commitment at a new random position, which the hiding property of KZG should prevent them from doing. At the same time, aggregation of openings can still be done using a random linear combination, just as with regular KZG. Aggregated openings can be verified given the list of commitments by verifying that each individual commitment is indeed valid and then using the linear combination of the KZG part of the commitments to verify the aggregated opening. Finally, we note that while our commitment is conceptually simple, the proof that it provides simulation-extractability is far from it.

On the Necessity of Randomness Beacons. Throughout our paper, we assume that all parties have access to a randomness beacon. It is sensible to ask how necessary this assumption is. Intuitively, we would like our lotteries to ensure that no party can predict when they will win a lottery. For this to be feasible, there needs to be a source of entropy associated with each lottery execution, which is exactly what a randomness beacon provides. From a practical perspective, assuming the existence of a randomness beacon is also not too problematic, as they are deployed and running already. In the context of Ethereum, for example, the randomness beacon is known as Randao[2].

On Simulation-Based Security. In this work, we have chosen to define aggregatable lotteries through ideal functionalities in the UC framework. An alternative approach could have been to give game-based definitions. We believe that ideal functionalities are the right approach here for two reasons. Firstly, it is not at all clear what equivalent "clean" game-based notions would look like. Designing game-based notions that, for example, ensure that the adversary does not win disproportionally often or that the winning probabilities in each lottery are independent is non-trivial and would result in complex definitions. This would then make using our primitive in other contexts more cumbersome. Secondly, we would like to guarantee that our lotteries remain secure, even if composed arbitrarily with other protocols. Ideal functionalities in the UC model provide us with this guarantee, whereas game-based notions do not in general.

Parallels with Multi-signatures. On a conceptual level, our contribution in this work has some strong parallels to multi-signatures [29,36]. These allow for aggregating many individual signatures for the same message into one short aggregate signature. Using multi-signatures one can significantly reduce the on-chain storage in blockchains like Ethereum, as each block only needs to store a small value, which simultaneously vouches for many different signers having approved the block's contents. Similarly, our aggregate lotteries allow for storing a short digest, which simultaneously vouches for all elected committee members within one election. Apart from our technical realization of such lotteries, we

[2] https://eth2book.info/capella/part2/building_blocks/randomness/.

view our conceptual idea of compressing this lotteries as one of our important contributions.

1.4 Paper Organization

The main part of the paper is organized as follows. In Sect. 3, we introduce syntax and game-based security notions for aggregatable vector commitments. We also present our construction of this primitive. Then, in Sect. 4 we define aggregatable lotteries in the UC framework, present our construction from any aggregatable vector commitment. We show UC security of our lottery assuming the vector commitment meets the game-based notions we have defined. Finally, in Sect. 5, we discuss practical aspects and the efficiency of our construction.

2 Preliminaries

In this section, we fix notation and recall relevant cryptographic preliminaries.

Notation. For a finite set S, writing $s \leftarrow_\$ S$ means that s is sampled uniformly at random from S. For a probabilistic algorithm $\mathcal{A}$, we write $s := \mathcal{A}(x;\rho)$ to state that $\mathcal{A}$ is run on input x with random coins ρ, and the result is assigned to the variable s. If the coins ρ are sampled uniformly at random, we write $s \leftarrow \mathcal{A}(x)$. If we write $s \in \mathcal{A}(x)$, we mean that there are random coins such that when $\mathcal{A}$ is run on input x with these random coins, it outputs s. The security parameter λ is given implicitly to all algorithms (in unary). We denote the running time of an algorithm $\mathcal{A}$ by $\mathbf{T}(\mathcal{A})$. We use standard cryptographic notions, e.g., PPT, negligible. We define $[L] := \{1,\ldots,L\} \subseteq \mathbb{N}$. We let $\mathcal{B}(p)$ denote a Bernoulli distribution with $\Pr[b=1]=p$ for b sampled from $\mathcal{B}(p)$ (written as $b \leftarrow \mathcal{B}(p)$).

Pairings and Assumptions. We rely on the ℓ-DLOG assumption and the ℓ-SDH assumption [5,30]. For this and the remainder of this paper, let PGGen be an algorithm that on input 1^λ outputs the description of prime order groups $\mathbb{G}_1$, $\mathbb{G}_2, \mathbb{G}_T$ of order p, generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, and the description of a pairing, i.e., a non-degenerate bilinear map $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ for which $e(g_1, g_2)$ is a generator of $\mathbb{G}_T$. That is, PGGen outputs $\mathsf{par} = (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, p, e)$. Then, informally, the ℓ-DLOG assumption states that it is hard to output α given $(g_1^{\alpha^i})_{i=1}^{\ell}, g_2^{\alpha}$ for a random $\alpha \leftarrow_\$ \mathbb{Z}_p$, and the ℓ-SDH assumption states that it is hard to output $(c, g_1^{1/(\alpha+c)})$ for some c on the same input. Clearly, ℓ-DLOG is implied by ℓ-SDH.

Definition 1 (ℓ-DLOGAssumption). *We say that the ℓ-DLOG assumption holds relative to PGGen, if for any PPT algorithm $\mathcal{A}$, the following advantage is negligible:*

$$\mathsf{Adv}_{\mathcal{A},\mathsf{PGGen}}^{\ell\text{-}\mathsf{DLOG}}(\lambda) := \Pr\left[\mathcal{A}(\mathsf{par},\mathsf{In}) = \alpha \;\middle|\; \begin{array}{l} \mathsf{par} \leftarrow \mathsf{PGGen}(1^\lambda), \\ \alpha \leftarrow_\$ \mathbb{Z}_p,\ \mathsf{In} := ((g_1^{\alpha^i})_{i=1}^{\ell}, g_2^{\alpha}) \end{array}\right].$$

Definition 2 (ℓ-SDHAssumption). *We say that the ℓ-SDH assumption holds relative to PGGen, if for any PPT algorithm $\mathcal{A}$, the following advantage is negligible:*

$$\mathsf{Adv}^{\ell\text{-}\mathsf{SDH}}_{\mathcal{A},\mathsf{PGGen}}(\lambda) := \Pr\left[\begin{array}{l} \exists c \in \mathbb{Z}_q \setminus \{-\alpha\} : \\ \mathcal{A}(\mathsf{par}, \mathsf{In}) = \left(c, g_1^{1/(\alpha+c)}\right) \end{array} \middle| \begin{array}{l} \mathsf{par} \leftarrow \mathsf{PGGen}(1^\lambda), \\ \alpha \leftarrow_{\$} \mathbb{Z}_p, \\ \mathsf{In} := ((g_1^{\alpha^i})_{i=1}^{\ell}, g_2^{\alpha}) \end{array}\right].$$

Universal Composability. We define an ideal functionality for aggregatable lotteries and prove security of our construction in the universal composability (UC) framework [13] in the presence of static corruptions. Our construction relies on synchronous broadcast and a synchronous randomness beacon. We include definitions of the UC functionalities, protocols, and the security proof in our full version [21]. When specifying functionalities and simulators, we write $\mathsf{msg} \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{port}$ to denote the event of receiving the message msg on the (possibly emulated) port port. Correspondingly, we use $\mathsf{port} \overset{\mathsf{send}}{\hookleftarrow} \mathsf{msg}$ to denote the sending of the message msg on the (possibly emulated) port port.

Random Oracle Model. For some of our proofs, we use the (programmable) random oracle model (ROM) [3]. To recall, in the ROM, hash functions are modeled by oracles implementing perfectly random functions via lazy sampling. For our UC proof, we use the standard ROM, which is sometimes known as the local ROM as opposed to the global ROM [12].

Algebraic Group Model. For some of our proofs and extractors, we leverage the algebraic group model (AGM) [24]. In this model, we only consider so called algebraic algorithms. This means that whenever such an algorithm outputs a group element Y in some cyclic group $\mathbb{G}$ of prime order p, it also outputs a so called algebraic representation, which is a vector $(c_1, \ldots, c_k) \in \mathbb{Z}_p^k$ such that $Y = \prod_{i=1}^k X_i^{c_i}$. Here, $X_1, \ldots, X_k$ are all group elements that the algorithm received so far. We emphasize that we analyze the game-based security of some of our building blocks in the AGM, and then use this security in a black-box manner for our UC proof.

3 Aggregatable Vector Commitments

In this section, we define and instantiate a special class of vector commitments that we will use to construct aggregatable lotteries.

3.1 Syntax of Our Vector Commitments

A vector commitment allows a party to commit to a vector $\mathbf{m} \in \mathcal{M}^\ell$ over some alphabet $\mathcal{M}$, resulting in a commitment com. Later, the committer can open com at any position $i \in [\ell]$ by revealing $\mathbf{m}_i$ and a corresponding opening (proof) τ. One can then publicly verify the pair $(\mathbf{m}_i, \tau)$ with respect to com and i.

Our definition of vector commitments is special in two ways. First, it should be possible to publicly aggregate several openings for different commitments with respect to the same position. Precisely, we require the existence of an algorithm $\mathsf{Aggregate}$ that takes a list of L openings $\tau_j, j \in [L]$ (all for the same position $i \in [\ell]$) and outputs an aggregated opening τ. One can then verify τ with respect to a list of L commitments. For non-triviality, the aggregated τ should ideally be as large as one single τ_j. Note that a similar aggregation feature for openings of different commitments has been defined in [27]. The second non-standard part of our definition is that we explicitly model an algorithm VerCom that verifies whether commitments (not openings) are well-formed. For our security notions, this will be convenient.

Definition 3 (Vector Commitment Scheme). *A vector commitment scheme (VC) is a tuple* $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$ *of PPT algorithms with the following syntax:*

- $\mathsf{Setup}(1^\lambda, 1^\ell) \to \mathsf{ck}$ *takes as input the security parameter and a message length* ℓ*, and outputs a commitment key* ck*. We assume that* ck *specifies a message alphabet* $\mathcal{M}$*, opening space* $\mathcal{T}$*, and commitment space* $\mathcal{C}$*.*
- $\mathsf{Com}(\mathsf{ck}, \mathbf{m}) \to (\mathsf{com}, St)$ *takes as input a commitment key* ck *and a vector* $\mathbf{m} \in \mathcal{M}^\ell$*, and outputs a commitment* $\mathsf{com} \in \mathcal{C}$ *and a state* St*.*
- $\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) \to b$ *is deterministic, takes as input a commitment key* ck *and a commitment* com*, and outputs a bit* $b \in \{0, 1\}$*.*
- $\mathsf{Open}(\mathsf{ck}, St, i) \to \tau$ *takes as input a commitment key* ck*, a state* St*, and an index* $i \in [\ell]$*, and outputs an opening* $\tau \in \mathcal{T}$*.*
- $\mathsf{Aggregate}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, (\tau_j)_{j=1}^L) \to \tau$ *is deterministic, takes as input a commitment key* ck*, an index* $i \in [\ell]$*, a list of commitments* $\mathsf{com}_j \in \mathcal{C}$*, a list of symbols* $m_j \in \mathcal{M}$*, and a list of openings* $\tau_j \in \mathcal{T}$*, and outputs an opening* $\tau \in \mathcal{T}$*.*
- $\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau) \to b$ *is deterministic, takes as input a commitment key* ck*, an index* $i \in [\ell]$*, a list of commitments* $\mathsf{com}_j \in \mathcal{C}$*, a list of symbols* $m_j \in \mathcal{M}$*, and an opening* $\tau \in \mathcal{T}$*, and outputs a bit* $b \in \{0, 1\}$*.*

  Further, we require that the following properties holds:

  1. ***Commitment Completeness.*** *For any* $\ell \in \mathbb{N}$*, any* $\mathsf{ck} \in \mathsf{Setup}(1^\lambda, 1^\ell)$*, and any* $\mathbf{m} \in \mathcal{M}^\ell$*, we have*

$$\Pr\left[\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) = 1 \mid (\mathsf{com}, St) \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{m})\right] = 1.$$

  2. ***Opening Completeness.*** *For any* $\ell \in \mathbb{N}$*, any* $\mathsf{ck} \in \mathsf{Setup}(1^\lambda, 1^\ell)$*, any* $\mathbf{m} \in \mathcal{M}^\ell$*, and any* $i \in [\ell]$*, we have*

$$\Pr\left[\mathsf{Ver}(\mathsf{ck}, i, \mathsf{com}, \mathbf{m}_i, \tau) = 1 \;\middle|\; \begin{array}{l} (\mathsf{com}, St) \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{m}), \\ \tau \leftarrow \mathsf{Open}(\mathsf{ck}, St, j) \end{array}\right] = 1.$$

  3. ***Aggregation Completeness.*** *For any* $\ell \in \mathbb{N}$*, any* $\mathsf{ck} \in \mathsf{Setup}(1^\lambda, 1^\ell)$*, any* $L \in \mathbb{N}$*, any index* $i \in [\ell]$*, any list* $(m_j)_{j=1}^L \in \mathcal{M}^L$*, any list* $(\mathsf{com}_j)_{j=1}^L \in$

$\mathcal{C}^L$, any list $(\tau_j)_{j=1}^L \in \mathcal{T}^L$, we have

$$\begin{aligned} & \forall j \in [L] : \mathsf{Ver}(\mathsf{ck}, i, \mathsf{com}_j, m_j, \tau_j) = 1 \\ & \wedge\ \tau = \mathsf{Aggregate}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, (\tau_j)_{j=1}^L) \\ \implies \quad & \mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau) = 1. \end{aligned}$$

3.2 Simulation-Extractability

We define a strong simulation-extractability property for vector commitments. This property captures all properties that we will need for our UC proof, including both hiding and binding properties. Beyond that, it may be interesting in itself. The notion states that no adversary can distinguish between two games in which it is running, where one game models the real world, and the other game models an ideal world. The first property that our notion models is a strong form of *hiding*. Namely, we require that there is a way to set up the commitment key with a trapdoor, and this trapdoor allows a simulator to compute commitments without knowing the message, and later open these commitments at arbitrary positions to arbitrary symbols. This is modeled in our notion as follows. In the real world game, the adversary gets an honest commitment key. It also gets access to an oracle GetCom that outputs honestly computed commitments to messages of the adversary's choice. Another oracle GetOp provides openings for these commitments when the adversary asks for them. In the ideal world game, the commitment key is set up with a trapdoor and both commitments and openings are simulated. In addition to this hiding property, our notion models a strong form of *binding*. Namely, the adversary gets access to oracles SubCom and SubOp that allow it to submit commitments and openings for them. While the commitments and openings are simply verified in the real world game, there are additional checks in the ideal world game. Concretely, when the adversary submits a commitment com that is not output by GetCom, the game not only verifies it, but also tries to extract a preimage $(\mathbf{m}, \varphi)$ from it, such that $\mathbf{m}$ with randomness φ commits to com. If this extraction fails but com verifies, SubCom outputs 0 in the ideal world game, whereas it would output 1 in the real world game. In other words, indistinguishability of the games ensures that we can always extract preimages of commitments. In addition, our notion ensures that openings are consistent: (1) whatever we extracted in SubCom is consistent with any valid opening that the adversary submits later, and (2) if the adversary opens a commitment output by GetCom($\mathbf{m}$) at position i, then (2a) it opens to the respective $\mathbf{m}_i$, and (2b) it queried GetOp for this commitment at position i before. Our notion ensures this because in the ideal game, SubOp outputs 0 if one of the inconsistencies (1, 2a, 2b) occurs, whereas in the real game the output of SubOp only depends on whether the opening verifies.

Definition 4 (Simulation-Extractability of VC). *Consider a vector commitment scheme* $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$. *For any algorithm $\mathcal{A}$, any $\ell \in \mathbb{N}$, any algorithm* Ext, *and any triple of algorithms*

$\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$, *consider the game* ℓ-$\mathbf{SIM}$-$\mathbf{EXT}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda)$ *and the game* ℓ-$\mathbf{SIM}$-$\mathbf{EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}(\lambda)$ *defined in Fig. 1. We say that* VC *is simulation-extractable, if there are PPT algorithms* Ext *and* $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$ *such that for any polynomial* $\ell \in \mathbb{N}$ *and any PPT algorithm* $\mathcal{A}$*, the following advantage is negligible:*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{sim\text{-}ext}}_{\mathcal{A},\mathsf{VC},\mathsf{Sim},\mathsf{Ext},\ell}(\lambda) := \Big|&\Pr\left[\ell\text{-}\mathbf{SIM}\text{-}\mathbf{EXT}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda) \Rightarrow 1\right] \\ &-\Pr\left[\ell\text{-}\mathbf{SIM}\text{-}\mathbf{EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}(\lambda) \Rightarrow 1\right]\Big|.\end{aligned}$$

Then, we say VC *is simulation-extractable with extractor* Ext *and simulator* Sim*.*

Our simulation-extractability notion is well-suited for our UC proof. However, it models several distinct properties of the vector commitment simultaneously, which renders a direct proof of simulation-extractability complicated. Thus, we define three less complex security notions and show that in combination they imply simulation-extractability. The first notion, *equivocality*, is the hiding part of our simulation-extractability notion.

Definition 5 (Equivocal VC). *Consider a vector commitment scheme* $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$*. For any algorithm* $\mathcal{A}$*, any* $\ell \in \mathbb{N}$*, and any triple of algorithms* $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$ *consider the games* ℓ-$\mathbf{EQUIV}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},b}(\lambda)$ *for* $b \in \{0, 1\}$ *defined in Fig. 2. We say that* VC *is equivocal, if there are PPT algorithms* $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$ *such that for any polynomial* $\ell \in \mathbb{N}$ *and any PPT algorithm* $\mathcal{A}$*, the following advantage is negligible:*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{equiv}}_{\mathcal{A},\mathsf{VC},\mathsf{Sim},\ell}(\lambda) := \Big|&\Pr\left[\ell\text{-}\mathbf{EQUIV}^{\mathcal{A}}_{\mathsf{VC},0}(\lambda) \Rightarrow 1\right] \\ &-\Pr\left[\ell\text{-}\mathbf{EQUIV}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},1}(\lambda) \Rightarrow 1\right]\Big|.\end{aligned}$$

In this case, we say that VC *is equivocal with simulator* Sim*.*

The second and third notion focus on binding. Namely, the notion of *augmented extractability* states that we can extract preimages of commitments from any opening that the adversary outputs, even if it sees some honest commitments and openings. Notably, we do not allow the extractor to inspect the internal state of the oracles that output these honest commitments and openings, which is crucial for making this notion compose with equivocality.

Definition 6 (Augmented Extractability of VC). *Let* $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$ *denote a vector commitment scheme. For any algorithm* $\mathcal{A}$*, any algorithm* Ext*, any* $\ell \in \mathbb{N}$*, consider the game* ℓ-$\mathbf{AUG}$-$\mathbf{EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Ext}}(\lambda)$ *defined in Fig. 3. We say that* VC *satisfies augmented*

```
Game $\ell$-SIM-EXT$^{\mathcal{A}}_{\mathsf{VC},0}(\lambda)$
$c := 0,\ \mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$
$\mathrm{O}_G := (\textsc{GetCom}_0, \textsc{GetOp}_0)$
$\mathrm{O}_S := (\textsc{SubCom}_0, \textsc{SubOp}_0)$
return $\mathcal{A}^{\mathrm{O}_G, \mathrm{O}_S}(\mathsf{ck})$

Oracle $\textsc{GetCom}_0(\mathbf{m})$
$c := c + 1,\quad \mathsf{Msgs}[c] := \mathbf{m}$
$(\mathsf{com}, St) \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{m})$
$\mathsf{Coms}[c] := \mathsf{com},\quad \mathsf{St}[c] := St$
$\mathsf{Ops}[c] := \emptyset$
return $\mathsf{com}$

Oracle $\textsc{GetOp}_0(k, i)$
if $\mathsf{Coms}[k] = \bot$ : return $\bot$
if $i \in \mathsf{Ops}[k]$ : return $\bot$
$\mathsf{Ops}[k] := \mathsf{Ops}[k] \cup \{i\}$
$\tau \leftarrow \mathsf{Open}(\mathsf{ck}, \mathsf{St}[k], i)$
return $\tau$

Oracle $\textsc{SubCom}_0(\mathsf{com})$
if $\exists k$ s.t. $\mathsf{Coms}[k] = \mathsf{com}$ :
    return $0$
if $\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) = 0$ :
    return $0$
$\mathsf{Sub} := \mathsf{Sub} \cup \{\mathsf{com}\}$
return $1$
```

```
Game $\ell$-SIM-EXT$^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},\mathsf{Ext},1}(\lambda)$
$c := 0,\ (\mathsf{ck}, \mathsf{td}) \leftarrow \mathsf{TSetup}(1^\lambda, 1^\ell)$
$\mathrm{O}_G := (\textsc{GetCom}_1, \textsc{GetOp}_1)$
$\mathrm{O}_S := (\textsc{SubCom}_1, \textsc{SubOp}_1)$
return $\mathcal{A}^{\mathrm{O}_G, \mathrm{O}_S}(\mathsf{ck})$

Oracle $\textsc{GetCom}_1(\mathbf{m})$
$c := c + 1,\quad \mathsf{Msgs}[c] := \mathbf{m}$
$(\mathsf{com}, St) \leftarrow \mathsf{TCom}(\mathsf{ck})$
$\mathsf{Coms}[c] := \mathsf{com},\quad \mathsf{St}[c] := St$
$\mathsf{Ops}[c] := \emptyset$
return $\mathsf{com}$

Oracle $\textsc{GetOp}_1(k, i)$
if $\mathsf{Coms}[k] = \bot$ : return $\bot$
if $i \in \mathsf{Ops}[k]$ : return $\bot$
$\mathsf{Ops}[k] := \mathsf{Ops}[k] \cup \{i\}$
$\tau \leftarrow \mathsf{TOpen}(\mathsf{td}, \mathsf{St}[k], i, \mathsf{Msgs}[k]_i)$
return $\tau$

Oracle $\textsc{SubCom}_1(\mathsf{com})$
if $\exists k$ s.t. $\mathsf{Coms}[k] = \mathsf{com}$ :
    return $0$
if $\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) = 0$ :
    return $0$
$(\mathbf{m}, \varphi) \leftarrow \mathsf{Ext}(\mathsf{td}, \mathsf{com})$
$(\mathsf{com}', St) := \mathsf{Com}(\mathsf{ck}, \mathbf{m}; \varphi)$
if $\mathsf{com}' \neq \mathsf{com}$ : return $0$
$\mathsf{MsgsExt}[\mathsf{com}] := \mathbf{m}$
$\mathsf{Sub} := \mathsf{Sub} \cup \{\mathsf{com}\}$
return $1$
```

```
Oracle $\textsc{SubOp}_b(i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau)$
if $b = 1$ : for $j \in [L]$ :
    if $\mathsf{com}_j \in \mathsf{Sub} \wedge m_j \neq \mathsf{MsgsExt}[\mathsf{com}]_i$ : return $0$
    if $\exists k$ s.t. $\mathsf{com}_j = \mathsf{Coms}[k] \wedge i \in \mathsf{Ops}[k] \wedge m_j \neq \mathsf{Msgs}[k]_i$ : return $0$
    if $\exists k$ s.t. $\mathsf{com}_j = \mathsf{Coms}[k] \wedge i \notin \mathsf{Ops}[k]$ : return $0$
return $\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau)$
```

Fig. 1. The simulation-extractability games ℓ-**SIM-EXT** for a vector commitment $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$, an adversary $\mathcal{A}$, an extractor Ext, and a simulator $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$. In the random oracle model, Ext gets as additional input the list of random oracle queries of $\mathcal{A}$. In the algebraic group model, Ext gets as additional input the algebraic representation of all group elements contained in the commitment com submitted by $\mathcal{A}$.

```
Game $\ell$-EQUIV$^{\mathcal{A}}_{\mathsf{VC},0}(\lambda)$
01 $c := 0$
02 $\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$
03 return $\mathcal{A}^{\textsc{GetCom}_0,\textsc{GetOp}_0}(\mathsf{ck})$

Game $\ell$-EQUIV$^{\mathcal{A}}_{\mathsf{VC},\mathsf{Sim},1}(\lambda)$
04 $c := 0$
05 $(\mathsf{ck}, \mathsf{td}) \leftarrow \mathsf{TSetup}(1^\lambda, 1^\ell)$
06 return $\mathcal{A}^{\textsc{GetCom}_1,\textsc{GetOp}_1}(\mathsf{ck})$
```

Fig. 2. The equivocality games ℓ-**EQUIV** for a vector commitment $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$, an adversary $\mathcal{A}$, and a simulator $\mathsf{Sim} = (\mathsf{TSetup}, \mathsf{TCom}, \mathsf{TOpen})$. Oracles $\textsc{GetCom}_b$ and $\textsc{GetOp}_b$ are as in Fig. 1.

extractability, if there is a PPT algorithm Ext *such that for any polynomial* $\ell \in \mathbb{N}$ *and any PPT algorithm* $\mathcal{A}$*, the following advantage is negligible:*

$$\mathsf{Adv}^{\text{aug-ext}}_{\mathcal{A},\mathsf{VC},\mathsf{Ext},\ell}(\lambda) := \Pr\left[\ell\text{-}\mathbf{AUG}\text{-}\mathbf{EXT}^{\mathcal{A}}_{\mathsf{VC},\mathsf{Ext}}(\lambda) \Rightarrow 1\right].$$

In this case, we say that VC *satisfies augmented extractability with extractor* Ext.

Augmented extractability states that we can extract some preimage of adversarially submitted commitments. It does not state that what we extract is consistent with whatever the adversary opens later. For that, we define *aggregation position-binding*. Intuitively, we want that any two lists of commitments and openings that an adversary outputs are consistent, i.e., if they share a commitment, then the opened symbols for that commitment are the same. It turns out that we can further simplify this by assuming that one of the lists contains exactly one honestly computed commitment (with potentially biased randomness).

Definition 7 (Aggregation Position-Binding of VC). *Let* $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$ *be a vector commitment scheme. For any algorithm* $\mathcal{A}$ *and any* $\ell \in \mathbb{N}$*, consider the game* ℓ-**A**-**POS**-**BIND**$^{\mathcal{A}}_{\mathsf{VC}}(\lambda)$ *defined in Fig. 4. We say that* VC *is aggregation position-binding, if for any polynomial* $\ell \in \mathbb{N}$ *and any PPT algorithm* $\mathcal{A}$*, the following advantage is negligible:*

$$\mathsf{Adv}^{\text{a-pos-bind}}_{\mathcal{A},\mathsf{VC},\ell}(\lambda) := \Pr\left[\ell\text{-}\mathbf{A}\text{-}\mathbf{POS}\text{-}\mathbf{BIND}^{\mathcal{A}}_{\mathsf{VC}}(\lambda) \Rightarrow 1\right].$$

Next, we show that our notions imply simulation-extractability.

Lemma 1 (Informal). *If a vector commitment is equivocal, aggregation position-binding, and satisfies augmented extractability, then it is simulation-extractable.*

We give the formal statement and proof in our full version [21]. Here, we sketch a proof: we need to show that the real game and the ideal game of simulation-extractability are indistinguishable. For that, we start with the real game. In a first step, we change the game by extracting from all commitments that the adversary submits via SubCom, and let the oracle return 0 if extraction does not yield a valid preimage. The games are indistinguishable by augmented extractability. Note that now oracle SubCom is as in the ideal game. In the

```
Game $\ell$-AUG-EXT$^{\mathcal{A}}_{\mathsf{VC},\mathsf{Ext}}(\lambda)$
01 $\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$
02 $\mathsf{com} \leftarrow \mathcal{A}^{\textsc{GetCom}_0, \textsc{GetOp}_0}(\mathsf{ck})$
03 $(\mathbf{m}, \varphi) \leftarrow \mathsf{Ext}(\mathsf{ck}, \mathsf{com}), \quad (\mathsf{com}', St) := \mathsf{Com}(\mathsf{ck}, \mathbf{m}; \varphi)$
04 if $\nexists k$ s.t. $\mathsf{Coms}[k] = \mathsf{com} \wedge \mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) = 1 \wedge \mathsf{com} \neq \mathsf{com}'$ : return 1
05 return 0
```

Fig. 3. The augmented extractability game ℓ-**AUG**-**EXT** for a vector commitment $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$, an extractor Ext, and an adversary $\mathcal{A}$. Oracles $\textsc{GetCom}_0$ and $\textsc{GetOp}_0$ are as in Fig. 1. In the random oracle model, Ext gets as additional input the list of random oracle queries of $\mathcal{A}$. In the algebraic group model, Ext gets as additional input the algebraic representation of all group elements contained in the commitment com submitted by $\mathcal{A}$. Notably, Ext does not share any internal state with the rest of the game.

```
Game $\ell$-A-POS-BIND$^{\mathcal{A}}_{\mathsf{VC}}(\lambda)$
01 $\mathsf{ck} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$
02 $(\mathbf{m}, \varphi, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau) \leftarrow \mathcal{A}(\mathsf{ck})$
03 $(\mathsf{com}, St) := \mathsf{Com}(\mathsf{ck}, \mathbf{m}; \varphi)$
04 if $\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau) = 0$ : return 0
05 if $\exists j^* \in [L]$ s.t. $\mathsf{com}_{j^*} = \mathsf{com} \wedge m_{j^*} \neq \mathbf{m}_i$ : return 1
06 return 0
```

Fig. 4. The aggregation position-binding game ℓ-**A**-**POS**-**BIND** for a vector commitment $\mathsf{VC} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{VerCom}, \mathsf{Open}, \mathsf{Aggregate}, \mathsf{Ver})$ and an adversary $\mathcal{A}$.

second step, we change oracle SubOp to be as in the ideal world game as well. The adversary can only distinguish this, if one of the three conditions on which the implementations of oracle SubOp in the real game and the ideal game differ occurs. It turns out that we can bound this probability using aggregation position-binding, see the full proof for more details. Now, it remains to change the implementation of oracles GetCom and GetOp to be as in the ideal game. This change can be done using equivocality of the commitment. Here, it is essential that we defined our extractor in an appropriate way, see Fig. 3: the extractor does not rely on any internals of the oracles GetCom and GetOp and just sees their input and output behavior. Otherwise, a reduction for this final change would not be able to run the extractor correctly.

3.3 Simulation-Extractable Vector Commitments from KZG

We present an instantiation of vector commitments with suitable properties based on the KZG commitment scheme [30]. We first recall the KZG commitment scheme [30]. Then, we modify it to get our vector commitment scheme with suitable properties.

- $\mathsf{KZG.Setup}(1^\lambda, 1^d) \to \mathsf{ck}$:
  1. Run $\mathsf{par} := (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, p, e) \leftarrow \mathsf{PGGen}(1^\lambda)$.
  2. Sample $\alpha \leftarrow_\$ \mathbb{Z}_p$ and $\beta \leftarrow_\$ \mathbb{Z}_p^*$, and set $h_1 := g_1^\beta \in \mathbb{G}_1$.
  3. Set $u_i := g_1^{\alpha^i}$ and $\hat{u}_i := h_1^{\alpha^i}$ for each $i \in \{0, \dots, d\}$. Set $R := g_2^\alpha$
  4. Return $\mathsf{ck} := (\mathsf{par}, h_1, R, (u_i)_{i=0}^d, (\hat{u}_i)_{i=0}^d)$.
- $\mathsf{KZG.Com}(\mathsf{ck}, f \in \mathbb{Z}_p[X]) \to (\mathsf{com}, St)$
  1. If the degree of f is larger than d, abort.
  2. Sample a polynomial $\hat{f} \in \mathbb{Z}_p[X]$ of degree d uniformly at random.
  3. Compute $\mathsf{com} = g_1^{f(\alpha)} \cdot h_1^{\hat{f}(\alpha)}$. Note that com can be computed efficiently.
  4. Return com and $St := (f, \hat{f})$.
- $\mathsf{KZG.Open}(\mathsf{ck}, St = (f, \hat{f}), z) \to \tau$
  1. Let $\psi := (f - f(z))/(X - z) \in \mathbb{Z}_p[X]$ and $\hat{\psi} := (\hat{f} - \hat{f}(z))/(X - z) \in \mathbb{Z}_p[X]$.
  2. Set $v := g_1^{\psi(\alpha)} \cdot h_1^{\hat{\psi}(\alpha)}$. Note that v can be computed efficiently.
  3. Return $\tau := (\hat{f}(z), v)$.
- $\mathsf{KZG.Ver}(\mathsf{ck}, \mathsf{com}, z, y, \tau = (\hat{y}, v)) \to b$
  1. If $e\left(\mathsf{com} \cdot g_1^{-y} \cdot h_1^{-\hat{y}}, g_2\right) = e\left(v, R \cdot g_2^{-z}\right)$, return $b = 1$. Else, return $b = 0$.

Let $\mathsf{H}\colon \{0,1\}^* \to \mathbb{Z}_p$ and $\mathsf{H}'\colon \{0,1\}^* \to \mathbb{Z}_p$ be random oracles. We now define the vector commitment scheme $\mathsf{VC}_{\mathsf{KZG}} = (\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Setup}, \mathsf{VC}_{\mathsf{KZG}}.\mathsf{Com}, \mathsf{VC}_{\mathsf{KZG}}.\mathsf{Open}, \mathsf{VC}_{\mathsf{KZG}}.\mathsf{Ver})$. Roughly, we use the linear properties of KZG to make aggregation work. To add non-malleability at the same time, we include an opening at a random position z_0 in the commitments. Typically, to commit to a vector of ℓ elements, one would work with polynomials of degree $d = \ell - 1$. As we give out one additional point $f(z_0)$ for non-malleability and still need hiding, it is natural to increase d by one to $d = \ell$. It turns out that for a technical reason in our extractability proof (see paragraph "Proof Strategy" in the proof of Lemma 4), we have to choose $d = \ell + 1$.

- $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Setup}(1^\lambda, 1^\ell) \to \mathsf{ck}$:
  1. Run $\mathsf{ck}_{\mathsf{KZG}} \leftarrow \mathsf{KZG.Setup}(1^\lambda, 1^d)$, where $d := \ell + 1$. The parameters specify message alphabet $\mathcal{M} := \mathbb{Z}_p$, opening space $\mathcal{T} := \mathbb{Z}_p \times \mathbb{G}_1$, and commitment space $\mathcal{C} := \mathbb{G}_1 \times \mathbb{Z}_p \times \mathcal{T}$.
  2. Let $\iota\colon [\ell] \to \mathbb{Z}_p$ be a fixed injection and $z_{\mathsf{out}} \in \mathbb{Z}_p$ such that 0 and z_{out} are not in the image of ι.
  3. Return $\mathsf{ck} := (\mathsf{ck}_{\mathsf{KZG}}, z_{\mathsf{out}}, \iota)$.
- $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Com}(\mathsf{ck}, \mathbf{m}) \to (\mathsf{com}, St)$
  1. Sample $\delta_0, \delta_1 \leftarrow_\$ \mathbb{Z}_p$ and let $f \in \mathbb{Z}_p[X]$ be the unique polynomial of degree $d := \ell + 1$ such that $f(0) = \delta_0$, $f(z_{\mathsf{out}}) = \delta_1$ and $f(\iota(i)) = \mathbf{m}_i$ for all $i \in [\ell]$.
  2. Run $(\mathsf{com}_{\mathsf{KZG}}, St) \leftarrow \mathsf{KZG.Com}(\mathsf{ck}, f \in \mathbb{Z}_p[X])$.
  3. Compute $z_0 := \mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$ and set $y_0 := f(z_0)$.
  4. Run $\tau_0 \leftarrow \mathsf{KZG.Open}(\mathsf{ck}_{\mathsf{KZG}}, St, z_0)$.
  5. Return $\mathsf{com} := (\mathsf{com}_{\mathsf{KZG}}, y_0, \tau_0)$ and St.
- $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) \to b$
  1. Parse $\mathsf{com} = (\mathsf{com}_{\mathsf{KZG}}, y_0, \tau_0)$ and set $z_0 := \mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$.
  2. Return $\mathsf{KZG.Ver}(\mathsf{ck}_{\mathsf{KZG}}, \mathsf{com}_{\mathsf{KZG}}, z_0, y_0, \tau_0)$.

- $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Open}(\mathsf{ck}, St, i) \to \tau$
  1. Return $\mathsf{KZG.Open}(\mathsf{ck}_{\mathsf{KZG}}, St, \iota(i))$.
- $\mathsf{Aggregate}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^{L}, (m_j)_{j=1}^{L}, (\tau_j)_{j=1}^{L}) \to \tau$
  1. For each $j \in [L]$, parse $\tau_j = (\hat{y}_j, v_j) \in \mathbb{Z}_p \times \mathbb{G}_1$.
  2. Set $\xi := \mathsf{H}'(i, (\mathsf{com}_j)_{j=1}^{L}, (m_j)_{j=1}^{L})$.
  3. Set $\hat{y} := \sum_{j=1}^{L} \xi^{j-1}\hat{y}_j$ and $v := \prod_{j=1}^{L} v_j^{\xi^{j-1}}$.
  4. Return $\tau = (\hat{y}, v)$.
- $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^{L}, (m_j)_{j=1}^{L}, \tau) \to b$
  1. For each $j \in [L]$, parse $\mathsf{com}_j = (\mathsf{com}_{\mathsf{KZG},j}, y_{0,j}, \tau_{0,j}) \in \mathbb{G}_1 \times \mathbb{Z}_p \times \mathcal{T}$.
  2. Set $\xi := \mathsf{H}'(i, (\mathsf{com}_j)_{j=1}^{L}, (m_j)_{j=1}^{L})$.
  3. Compute $m := \sum_{j=1}^{L} \xi^{j-1} m_j$ and $\mathsf{com} := \prod_{j=1}^{L} \mathsf{com}_{\mathsf{KZG},j}^{\xi^{j-1}}$.
  4. Return $\mathsf{KZG.Ver}(\mathsf{ck}_{\mathsf{KZG}}, \mathsf{com}, \iota(i), m, \tau)$.

In the following, we show that $\mathsf{VC}_{\mathsf{KZG}}$ satisfies equivocality, aggregation position-binding, and augmented extractability. Simulation-extractability then follows from Lemma 1.

Lemma 2 (Informal). *Let* $\mathsf{H}\colon \{0,1\}^* \to \mathbb{Z}_p$ *be a random oracle. Then,* $\mathsf{VC}_{\mathsf{KZG}}$ *is equivocal.*

We provide the formal statement and proof in our full version [21]. Intuitively, the simulator sets $\mathsf{com}_{\mathsf{KZG}}$ to be a random group element, samples the polynomials f and $\hat{f}$ in a lazy way, and computes openings on the fly using knowledge of the trapdoor α and the equation $v = \left(\mathsf{com}_{\mathsf{KZG}} \cdot g_1^{-y} h_1^{-\hat{y}}\right)^{\frac{1}{\alpha - z}}$. To make the formal argument work, we need to carry out the changes in the correct order and pay attention to the degrees of the polynomials.

Lemma 3. *If the* d-DLOG *assumption holds relative to* PGGen *and* $\mathsf{H}'\colon \{0,1\}^* \to \mathbb{Z}_p$ *is modeled as a random oracle, then* $\mathsf{VC}_{\mathsf{KZG}}$ *is aggregation position-binding in the algebraic group model. Concretely, for any polynomial* $\ell \in \mathbb{N}$ *and any algebraic PPT algorithm* $\mathcal{A}$ *that makes at most* $Q_{\mathsf{H}'}$ *queries to random oracle* H', *there are PPT algorithms* $\mathcal{B}_1, \mathcal{B}_2$ *with* $\mathbf{T}(\mathcal{B}_1) \approx \mathbf{T}(\mathcal{B}_2) \approx \mathbf{T}(\mathcal{A})$ *and*

$$\mathsf{Adv}^{\mathsf{a\text{-}pos\text{-}bind}}_{\mathcal{A},\mathsf{VC}_{\mathsf{KZG}},\ell}(\lambda) \leq 2 \cdot \mathsf{Adv}^{\mathsf{1\text{-}DLOG}}_{\mathcal{B}_1,\mathsf{PGGen}}(\lambda) + 2 \cdot \mathsf{Adv}^{(\ell+1)\text{-}\mathsf{DLOG}}_{\mathcal{B}_2,\mathsf{PGGen}}(\lambda) + \frac{Q_{\mathsf{H}'} L_{max}}{p},$$

Proof. We first recall the aggregation position-binding game for an algebraic adversary $\mathcal{A}$ and a dimension ℓ to fix some notation. Set $d := \ell + 1$ as in the scheme. First, a commitment key $\mathsf{ck} = (\mathsf{ck}_{\mathsf{KZG}}, z_{\mathsf{out}}, \iota)$ is generated, where ι is a mapping from $[\ell]$ to $\mathbb{Z}_p$ and $\mathsf{ck}_{\mathsf{KZG}} = (\mathsf{par}, h_1, R, (u_i)_{i=0}^{d}, (\hat{u}_i)_{i=0}^{d})$ is a commitment key for the KZG polynomial commitment, with group parameters $\mathsf{par} = (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, p, e)$. That is, $h_1 = g_1^{\beta}$ for some $\beta \in \mathbb{Z}_p$, and there is some $\alpha \in \mathbb{Z}_p$ such that $u_i = g_1^{\alpha^i}$ and $\hat{u}_i = h_1^{\alpha^i}$ for each $i \in \{0, \ldots, d\}$. Further, $R = g_2^{\alpha}$. Then, when $\mathcal{A}$ terminates, it outputs the following:

- A message $\mathbf{m} \in \mathbb{Z}_p^\ell$ and randomness φ. Here φ has the form $\varphi = (\delta_0, \delta_1, \hat{f}') \in \mathbb{Z}_p \times \mathbb{Z}_p \times \mathbb{Z}_p[X]$, where $\hat{f}'$ is of degree ℓ. Based on this output, the aggregation position-binding game honestly computes a commitment com. More concretely, let $f' \in \mathbb{Z}_p[X]$ be the polynomial of degree $d = \ell + 1$ with $f'(0) = \delta_0$, $f'(z_{\mathsf{out}}) = \delta_1$, and $f'(\iota(i)) = \mathbf{m}_i$ for every $i \in [\ell]$. Then, the commitment com has the form $\mathsf{com} = (\mathsf{com}_{\mathsf{KZG}}, y_0, \tau_0)$, where $\mathsf{com}_{\mathsf{KZG}} = g_1^{f'(\alpha)} h_1^{\hat{f}'(\alpha)}$.
- An index $i^* \in [\ell]$. We will denote $z := \iota(i^*)$. Further, we will denote by ψ', $\hat{\psi}' \in \mathbb{Z}_p[X]$ the polynomials

$$\psi' := \frac{f' - f'(z)}{X - z}, \quad \hat{\psi}' := \frac{\hat{f}' - \hat{f}'(z)}{X - z}$$

  as in an honest KZG opening for f' at position z. The game can efficiently compute these polynomials.
- Lists $(\mathsf{com}_j)_{j=1}^L$ and $(m_j)_{j=1}^L$, and an opening $\tau = (\hat{y}, v) \in \mathbb{Z}_p \times \mathbb{G}_1$. Concretely, each com_j has the form $\mathsf{com}_j = (\mathsf{com}_{\mathsf{KZG},j}, y_{0,j}, \tau_{0,j})$, where $\mathsf{com}_{\mathsf{KZG},j} \in \mathbb{G}_1$. As $\mathcal{A}$ is algebraic, it also outputs an algebraic representation for each $\mathsf{com}_{\mathsf{KZG},j}$ and for τ. Due to the structure of the commitment key, this is equivalent to saying that $\mathcal{A}$ outputs polynomials $f_j, \hat{f}_j \in \mathbb{Z}_p[X]$ and $\psi, \hat{\psi} \in \mathbb{Z}_p[X]$ all of degree at most $d = \ell + 1$ such that

$$\tau = g_1^{\psi(\alpha)} \cdot h_1^{\hat{\psi}(\alpha)} \wedge \forall j \in [L] : \mathsf{com}_{\mathsf{KZG},j} = g_1^{f_j(\alpha)} \cdot h_1^{\hat{f}_j(\alpha)}.$$

We denote the event that $\mathcal{A}$ breaks aggregation position-binding by Win. Assuming that Win occurs, we know the following: There is an index $j^* \in [L]$ such that $\mathsf{com}_{j^*} = \mathsf{com}$ and $m_{j^*} \neq \mathbf{m}_{i^*}$. In particular, this means that $\mathsf{com}_{\mathsf{KZG}} = \mathsf{com}_{\mathsf{KZG},j^*}$ and $m_{j^*} \neq f'(z)$. We have $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{Ver}(\mathsf{ck}, i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L, \tau) = 1$. In particular, by reading the verification equation in the exponent, we have

$$\sum_{j=1}^{L} \xi^{j-1}(f_j(\alpha) + \beta \hat{f}_j(\alpha) - m_j) - \beta \hat{y} = (\psi(\alpha) + \beta \hat{\psi}(\alpha))(\alpha - z)$$

for $\xi := \mathsf{H}'(i, (\mathsf{com}_j)_{j=1}^L, (m_j)_{j=1}^L)$. Defining polynomials $f := \sum_{j=1}^L f_j \xi^{j-1} \in \mathbb{Z}_p[X]$ and $\hat{f} := \sum_{j=1}^L \hat{f}_j \xi^{j-1} \in \mathbb{Z}_p[X]$, and the element $m := \sum_{j=1}^L m_j \xi^{j-1}$ simplifies this equation to

$$f(\alpha) + \beta \hat{f}(\alpha) - m - \beta \hat{y} = (\psi(\alpha) + \beta \hat{\psi}(\alpha))(\alpha - z).$$

By construction of $f', \hat{f}'$ and $\psi', \hat{\psi}'$, we also have

$$f'(\alpha) + \beta \hat{f}'(\alpha) - f'(z) - \beta \hat{f}'(z) = (\psi'(\alpha) + \beta \hat{\psi}'(\alpha))(\alpha - z).$$

Subtracting the two equations, we get our *core equation*, namely,

$$\begin{aligned} & f(\alpha) - f'(\alpha) + \beta(\hat{f}(\alpha) - \hat{f}'(\alpha)) - (m - f'(z)) - \beta(\hat{y} - \hat{f}'(z)) \\ = \; & (\psi(\alpha) - \psi'(\alpha) + \beta(\hat{\psi}(\alpha) - \hat{\psi}'(\alpha)))(\alpha - z). \end{aligned}$$

Our goal will be to simplify the structure of this core equation. To do so, our first step is to eliminate the terms containing β. We define the following event:

– Event $\mathsf{Complex}$: This event occurs, if $\eta := \hat{f}(\alpha) - \hat{f}'(\alpha) - (\hat{y} - \hat{f}'(z)) - (\hat{\psi}(\alpha) - \hat{\psi}'(\alpha))(\alpha - z) \neq 0$.

Claim. There is a PPT algorithm $\mathcal{B}$ with $\Pr[\mathsf{Win} \wedge \mathsf{Complex}] \leq \mathsf{Adv}^{\mathsf{1\text{-}DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda)$.

Proof of Claim. Reduction $\mathcal{B}$ is as follows. It gets as input the group parameters, the generator g_1 and the element $h_1 = g_1^{\beta}$. We show that it can simulate the game for $\mathcal{A}$ and compute β if $\mathsf{Win} \wedge \mathsf{Complex}$ occurs. For that, $\mathcal{B}$ first samples $\alpha \leftarrow_{\$} \mathbb{Z}_p$ and computes the commitment key ck as in algorithm Setup. Observe that for that, β is not needed. Now, if Win occurs, then we know that the core equation holds. For convenience, we group together the β-terms in our core equation, getting

$$\beta \cdot \eta = f'(\alpha) - f(\alpha) + (m - f'(z)) + (\psi(\alpha) - \psi'(\alpha))(\alpha - z).$$

Clearly, if $\mathsf{Complex}$, then reduction $\mathcal{B}$ can compute β by multiplying the right hand-side with η^{-1}.

From now on, we condition on $\neg\mathsf{Complex}$. In other words, we assume that $\eta = 0$, which implies that the *simplified core equation*

$$f(\alpha) - m - (f'(\alpha) - f'(z)) = (\psi(\alpha) - \psi'(\alpha))(\alpha - z)$$

holds. Our goal will be to show that if this equation holds, we can build a reduction breaking the d-DLOG assumption. For that, we define the following events:

– Event NoColl : This event occurs, if $f'(\alpha) \neq f_{j^*}(\alpha)$.
– Event Ambig : This event occurs, if $f' \neq f_{j^*}$ over $\mathbb{Z}_p[X]$.
– Event $\mathsf{AggFail}$: This event occurs, if $m = f(z)$.

In the next claims, we bound the probability that one of these event occurs. Informally, if NoColl occurs, then one can use $\mathsf{com}_{\mathsf{KZG}} = \mathsf{com}_{\mathsf{KZG},j^*}$ solve for β to break DLOG. If Ambig occurs but NoColl does not, then we can find α efficiently as a root of the non-zero polynomial $f' - f_{j^*}$. We will bound the case that $\mathsf{AggFail}$ occurs using a statistical argument using the fact that ξ is chosen after the f_j and m_j are fixed.

Claim. There is a PPT algorithm $\mathcal{B}$ with $\Pr[\mathsf{Win} \wedge \mathsf{NoColl}] \leq \mathsf{Adv}^{\mathsf{1\text{-}DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda)$.

Proof of Claim. Note that if Win occurs, then we have $f'(\alpha) + \beta\hat{f}'(\alpha) = f_{j^*}(\alpha) + \beta\hat{f}_{j^*}(\alpha)$, because $\mathsf{com}_{\mathsf{KZG}} = \mathsf{com}_{\mathsf{KZG},j^*}$. If NoColl occurs at the same time, then we know that $\hat{f}'(\alpha) - \hat{f}_{j^*}(\alpha) \neq 0$ and one can efficiently solve for β. It is not hard to turn that into a formal reduction that determines β given $h_1 = g_1^{\beta}$.

Claim. There is a PPT algorithm $\mathcal{B}$ with $\Pr[\mathsf{Ambig} \wedge \neg\mathsf{NoColl}] \leq \mathsf{Adv}^{d\text{-}\mathsf{DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda)$.

Proof of Claim. Note that if $\neg\mathsf{NoColl}$ occurs, then we have $f'(\alpha) \neq f_{j^*}(\alpha)$. If Ambig occurs at the same time, we know that α is a root of the non-zero polynomial $f' - f_{j^*}$, which has degree at most $d = \ell + 1$. Hence, α can be found in polynomial time by a reduction. We leave details to the reader.

Claim. We have $\Pr[\mathsf{Win} \wedge \mathsf{AggFail} \wedge \neg\mathsf{Ambig}] \leq Q_{\mathsf{H}'} L_{max}/p$.

Proof of Claim. By definition of m and f, event $\mathsf{AggFail}$ is equivalent to the equation

$$\sum_{j=1}^{L} m_j \xi^{j-1} = \sum_{j=1}^{L} f_j(z) \xi^{j-1}.$$

In other words, if $\mathsf{AggFail}$ occurs, then the polynomial

$$\zeta = \sum_{j=1}^{L} (f_j(z) - m_j) X^{j-1} \in \mathbb{Z}_p[X]$$

has a root at ξ. Observe that ζ has degree $L \leq L_{max}$ and is fixed before[3] ξ is chosen at random by the random oracle H'. Thus, for any fixed random oracle query where $\zeta \neq 0$, this event occurs with probability at most L_{max}/p. It remains to argue that $\zeta \neq 0$ if Win occurs and Ambig does not. This can be seen by observing that the j^*th coefficient of ζ is non-zero, i.e., $m_{j^*} \neq f_{j^*}(z)$. Namely, we know that $f' = f_{j^*}$ due to $\neg\mathsf{Ambig}$. Thus, if we had $m_{j^*} = f_{j^*}(z)$, then we had $m_{j^*} = f'(z)$, contradicting Win.

Concluding the Proof. To conclude the proof, we can now assume that Win occurs, but neither of the events defined above occurs. We come back to our simplified core equation. The equation tells us that α is a root of the polynomial Ψ of degree at most $d = \ell + 1$, which is given as

$$\Psi = f - m - (f' - f'(z)) - (\psi - \psi')(X - z) \in \mathbb{Z}_p[X].$$

Now, if we can argue that Ψ is non-zero, then one can efficiently find α based on $\mathcal{A}$'s output, leading to our final reduction. To argue that Ψ is non-zero, note that $\Psi = 0$ implies that

$$f - m = (f' - f'(z)) + (\psi - \psi')(X - z).$$

While the right hand-side is a multiple of $X - z$, the left hand-side is not, as we assume $\neg\mathsf{AggFail}$. Therefore, this equality can not hold, which means that Ψ is non-zero. In combination, setting $\mathsf{Bad} := \mathsf{NoColl} \vee \mathsf{Ambig} \vee \mathsf{AggFail} \vee \mathsf{Complex}$ we get a reduction $\mathcal{B}$ with

$$\Pr[\mathsf{Win} \wedge \neg\mathsf{Bad}] \leq \mathsf{Adv}^{d\text{-}\mathsf{DLOG}}_{\mathcal{B},\mathsf{PGGen}}(\lambda).$$

Lemma 4 (Informal). *If d-DLOG holds and $\mathsf{H}\colon \{0,1\}^* \rightarrow \mathbb{Z}_p$ is a random oracle, then $\mathsf{VC}_{\mathsf{KZG}}$ satisfies augmented extractability in the algebraic group model.*

[3] It could be that the adversary submitted a different algebraic representation to the random oracle. In this case, we just define the f_j's to be this representation.

We provide the formal statement and proof in our full version [21]. Here, we first recall the augmented extractability game to fix notation and define our extractor Ext. Then, we provide a proof intuition.

Game, Extractor and Notation. In the augmented extractability game, the adversary $\mathcal{A}$ gets as input a commitment key ck and access to a commitment oracle GetCom and an opening oracle GetOp. These are as follows:

- The commitment key ck has the form $\mathsf{ck} = (\mathsf{ck}_{\mathsf{KZG}}, z_{\mathsf{out}}, \iota)$ where $\iota\colon [\ell] \to \mathbb{Z}_p$ is injective and $\mathsf{ck}_{\mathsf{KZG}} = (\mathsf{par}, h_1, R, (u_i)_{i=0}^{d}, (\hat{u}_i)_{i=0}^{d})$ is a KZG commitment key with group parameters $\mathsf{par} = (\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, p, e)$. We have $h_1 = g_1^{\beta}$ for some random $\beta \in \mathbb{Z}_p$, and $u_i = g_1^{\alpha^i}$ and $\hat{u}_i = h_1^{\alpha^i}$ for each $i \in \{0, \dots, d\}$ for some random $\alpha \in \mathbb{Z}_p$. We also have $R = g_2^{\alpha}$.
- On input $\mathbf{m} \in \mathbb{Z}_p^{\ell}$, the commitment oracle returns a commitment com for $\mathbf{m}$. We use the subscript k to refer to the kth commitment returned by the oracle. That is, $\mathsf{com}_k = (\mathsf{com}_{\mathsf{KZG},k}, f_k(z_{k,0}), \tau_{k,0})$ is the kth commitment returned by the oracle, where $\mathsf{com}_{\mathsf{KZG},k} = g_1^{f_k(\alpha)} h_1^{\hat{f}_k(\alpha)}$ is a KZG commitment to a polynomial f_k of degree d with randomness $\hat{f}_k$, and $\tau_{k,0} = (\hat{f}_k(z_{k,0}), v_{k,0})$ is a KZG opening for f_k at position $z_{k,0} = \mathsf{H}(\mathsf{com}_{\mathsf{KZG},k})$ to value $f_k(z_{k,0})$. We denote the number of queries to GetCom by Q_C and assume without loss of generality that $Q_C \geq 1$.
- On input (k, i) such that com_k is defined, the opening oracle GetOp opens com_k at position i. To recall, such an opening is a KZG openings for commitment $\mathsf{com}_{\mathsf{KZG},k}$ at position $z_{k,i} := \iota(i)$. We denote the opening by $\tau_{k,i} = (\hat{f}_k(z_{k,i}), v_{k,i})$ for $v_{k,i} = g_1^{\psi_{k,i}(\alpha)} h_1^{\hat{\psi}_{k,i}(\alpha)}$, where $\psi_{k,i} = (f_k - f_k(z_{k,i}))/(X - z_{k,i}) \in \mathbb{Z}_p[X]$ and $\hat{\psi}_{k,i} := (\hat{f}_k - \hat{f}_k(z_{k,i}))/(X - z_{k,i})$. Without loss of generality, we can assume that for every commitment com_k returned by the commitment oracle, $\mathcal{A}$ queries the opening oracle for every $i \in [\ell]$, and thus it obtained all of these openings $\tau_{k,i}$ for $i \in \{0, \dots, \ell\}$.

When $\mathcal{A}$ terminates, it outputs a commitment outputs $\mathsf{com} = (\mathsf{com}_{\mathsf{KZG}}, y_0, \tau_0)$ with $\tau_0 = (\hat{y}_0, v_0)$. As $\mathcal{A}$ is algebraic, it also outputs the algebraic representation of all group elements in com. Due to the structure of ck and the group elements that $\mathcal{A}$ obtained from the commitment and opening oracles, we can assume that this representation is given by polynomials $f, \hat{f}, \psi, \hat{\psi} \in \mathbb{Z}_p[X]$ of degree $d = \ell + 1$ and lists of exponents $(w_k)_k, (r_k)_k$ and $(t_{k,i})_{k,i}, (s_{k,i})_{k,i}$ over $\mathbb{Z}_p$ such that

$$\mathsf{com}_{\mathsf{KZG}} = g_1^{f(\alpha)} \cdot h_1^{\hat{f}(\alpha)} \cdot \underbrace{\prod_{k=1}^{Q_C} \mathsf{com}_{\mathsf{KZG},k}^{w_k} \cdot \prod_{k=1}^{Q_C} \prod_{i=0}^{\ell} v_{k,i}^{t_{k,i}}}_{=:L},$$

$$v_0 = g_1^{\psi(\alpha)} \cdot h_1^{\hat{\psi}(\alpha)} \cdot \prod_{k=1}^{Q_C} \mathsf{com}_{\mathsf{KZG},k}^{r_k} \cdot \prod_{k=1}^{Q_C} \prod_{i=0}^{\ell} v_{k,i}^{s_{k,i}}.$$

Without loss of generality, we assume that $\mathcal{A}$ queried $\mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$, and it did so with the same algebraic representation for $\mathsf{com}_{\mathsf{KZG}}$ as the one that it outputs in the end. Given the output of the adversary, the extractor returns the message $\mathbf{m} \in \mathbb{Z}_p^\ell$ defined by $\mathbf{m}_i := f(\iota(i))$ for all $i \in [\ell]$ and the randomness $\varphi := (\delta_0, \delta_1, \hat{f})$, where $\delta_0 := f(0)$ and $\delta_1 := f(z_{\mathsf{out}})$. Now, $\mathcal{A}$ wins the game, if the following three properties hold:

- The commitment com is fresh, i.e., it was not output by the commitment oracle GetCom.
- Committing to $\mathbf{m}$ with randomness φ does not yield com. It is easy to see that this can only happen if $L \neq g_1^0$.
- The commitment com verifies, i.e., $\mathsf{VC}_{\mathsf{KZG}}.\mathsf{VerCom}(\mathsf{ck}, \mathsf{com}) = 1$. This is equivalent to saying that τ_0 is a valid KZG opening for $\mathsf{com}_{\mathsf{KZG}}$ at position $z_0 = \mathsf{H}(\mathsf{com}_{\mathsf{KZG}})$ to value y_0, i.e., that

$$e\left(\mathsf{com}_{\mathsf{KZG}} \cdot g_1^{-y_0} \cdot h_1^{-\hat{y_0}}, g_2\right) = e\left(v_0, g_2^{\alpha} \cdot g_2^{-z_0}\right).$$

Proof Strategy. In a preparatory phase (see $\mathbf{G}_0$ to $\mathbf{G}_3$), we rule out some simple bad events and simplify some equations. Namely, we rule out that there are collisions among the z's, e.g., that $z_0 = z_{k,i}$ for some i and k. We also rule out that α is equal to one of those z's. Further, we ensure that not only com is fresh, but instead $\mathsf{com}_{\mathsf{KZG}}$ is fresh. For that, we need to rule out that the adversary reuses a $\mathsf{com}_{\mathsf{KZG},k}$ with a different opening. We also expand the verification equation using the algebraic representation, and simplify it by ensuring that the exponent of h_1 is zero. Indeed, if this were not the case, one could compute the discrete logarithm β of $h_1 = g_1^\beta$. After this preparatory phase, our main argument follows (see $\mathbf{G}_4$ to $\mathbf{G}_7$). Namely, we show that from the adversary's output, a reduction can efficiently compute $f_{k^*}(z_0)$ for some k^*, while it only provided the $\ell + 1 = d$ evaluations $f_{k^*}(z_{k^*,i})$ for $i \in \{0, \ldots, \ell\}$ to the adversary. With this additional evaluation point $f_{k^*}(z_0)$, the reduction can compute f_{k^*} entirely. Roughly, this observation can be used as follows: The reduction guesses k^*, interprets a DLOG instance $X^* = g_1^{x^*}$ as $g_1^{f_{k^*}(\alpha)}$, and embeds it into the commitment $\mathsf{com}_{\mathsf{KZG},k^*}$. With the output of the adversary, it can efficiently recover f_{k^*} and therefore the discrete logarithm $x^* = f_{k^*}(\alpha)$. The details of the proof can be found in our full version [21].

4 Aggregatable Lotteries

In this section, we discuss how our lotteries are defined and how they can be constructed from our notion of vector commitments. Due to space constraints, we defer the formal description of our protocol as well as all corresponding security proofs to our full version [21].

4.1 Definition of Aggregatable Lotteries

We formally present our ideal functionality $\mathcal{F}_{\text{lottery}}(p, T)$ for *non-interactive aggregatable lotteries* in Figs. 5 and 6. It is parameterized by an upper bound on the winning probability p and the number of lotteries T. The probability p specifies how likely it is that a party wins in a lottery round. As described previously, our lotteries should intuitively prevent an adversary from winning the lotteries a disproportionate amount of times and the adversary should also not be able to tell which honest parties win the lotteries when. We do, however, allow the adversary to reduce its winning probability, i.e., the adversary can misbehave in a way that makes them win the lottery less often. We model this by allowing the adversary to specify their own winning probabilities for each lottery, capped at p, upon registration.

Our ideal functionality also relies on a party called the Croupier, which we did not mention so far. It is in charge of initiating lottery rounds and registering participants. Note that this model allows the adversary to corrupt Croupier, meaning that security is guaranteed even when the adversary can arbitrarily control the lottery, i.e., by registering players or initiating new lotteries.

Parties can be registered by Croupier via the Register interface. A lottery execution is run by Croupier via the NextLottery interface. Upon calling this interface, the functionality flips a biased coin for every currently registered party and stores whether they won the currently executed lottery. Parties can call the Participate interface to see whether they won a specific lottery. If they did, then they obtain a lottery ticket label, otherwise they receive nothing. Parties can call the Aggregate interface with a set of winning ticket labels and party identifiers to obtain an aggregate ticket label. Lastly, the Verify interface takes an aggregate ticket label and the corresponding winning parties' identifiers as input and checks whether the ticket is valid.

4.2 Our Construction

Let us now proceed with our construction of aggregatable lotteries from vector commitments. For that, let $\mathsf{VC} = (\mathsf{VC.Setup}, \mathsf{VC.Com}, \mathsf{VC.VerCom}, \mathsf{VC.Open}, \mathsf{VC.Aggregate}, \mathsf{VC.Ver})$ be a vector commitment scheme according to Definition 3. Let $T, k \in \mathbb{N}$ be arbitrary parameters. We construct a T-time aggregatable lottery with winning probability $p = 1/k$ using a random oracle $\mathsf{H}\colon \{0,1\}^* \to [k]$. The main idea is that each player commits to a vector $\mathbf{v} \in [k]^T$ upfront, and wins the ith lottery if and only if its *personal challenge* x is equal to $\mathbf{v}_i$. Crucially, the challenge x has to be independent for different players and over different lotteries, and should be unpredictable before the lottery seed lseed is known. Thus, we define $x := \mathsf{H}(\mathsf{pk}, \mathsf{pid}, i, \mathsf{lseed})$, where pid is the identifier of the player and pk is its public key, i.e., its commitment to $\mathbf{v}$.

Algorithms. To define our lottery protocol, we first define algorithms Setup, $\mathsf{Gen}, \mathsf{VerKey}, \mathsf{Participate}, \mathsf{Aggregate}, \mathsf{Ver}$ that will be used in our protocol:

– $\mathsf{Setup}(1^\lambda) \to \mathsf{par}$:
  1. Run $\mathsf{ck} \leftarrow \mathsf{VC.Setup}(1^\lambda, 1^T)$. Recall that ck defines message alphabet $\mathcal{M}$, opening space $\mathcal{T}$, and commitment space $\mathcal{C}$. We assume that $[k] \subseteq \mathcal{M}$.
  2. Set and return $\mathsf{par} := \mathsf{ck}$.

– $\mathsf{Gen}(\mathsf{par}) \to (\mathsf{pk}, \mathsf{sk})$:

   1. Sample $\mathbf{v} \leftarrow_{\$} [k]^T$ and run $(\mathsf{com}, St) \leftarrow \mathsf{VC.Com}(\mathsf{ck}, \mathbf{v})$.
   2. Set and return $\mathsf{pk} := \mathsf{com}$ and $\mathsf{sk} := (\mathbf{v}, St)$.
- $\mathsf{VerKey}(\mathsf{pk}) \rightarrow b$
   1. Return $b := \mathsf{VC.VerCom}(\mathsf{ck}, \mathsf{pk})$.
- $\mathsf{Participate}(t, \mathsf{lseed}, \mathsf{pid}, \mathsf{sk}) \rightarrow \mathsf{ticket}/\bot$:
   1. Set $x := \mathsf{H}(\mathsf{pk}, \mathsf{pid}, t, \mathsf{lseed})$. If $\mathbf{v}_i \neq x$, return $\bot$.
   2. Otherwise, set $\tau \leftarrow \mathsf{VC.Open}(\mathsf{ck}, St, i)$ and return $\mathsf{ticket} := \tau$.
- $\mathsf{Aggregate}(t, \mathsf{lseed}, (\mathsf{pid}_j, \mathsf{pk}_j)_{j=1}^L, (\mathsf{ticket}_j)_{j=1}^L) \rightarrow \mathsf{agg}$:
   1. For each $j \in [L]$, write $\mathsf{pk}_j = \mathsf{com}_j$ and $\mathsf{ticket}_j = \tau_j$.
   2. For each $j \in [L]$, set $x_j := \mathsf{H}(\mathsf{pk}_j, \mathsf{pid}_j, i, \mathsf{lseed})$.
   3. Return $\mathsf{agg} := \mathsf{VC.Aggregate}(\mathsf{ck}, t, (\mathsf{com}_j)_{j=1}^L, (x_j)_{j=1}^L, (\tau_j)_{j=1}^L)$.
- $\mathsf{Ver}(t, \mathsf{lseed}, (\mathsf{pid}_j, \mathsf{pk}_j)_{j=1}^L, \mathsf{agg} = \tau) \rightarrow b$:
   1. For each $j \in [L]$, write $\mathsf{pk}_j = \mathsf{com}_j$ and set $x_j := \mathsf{H}(\mathsf{pk}_j, \mathsf{pid}_j, t, \mathsf{lseed})$.
   2. Return $b := \mathsf{VC.Ver}(\mathsf{ck}, t, (\mathsf{com}_j)_{j=1}^L, (x_j)_{j=1}^L, \tau)$.

Protocol. We informally explain how to turn these algorithms into a protocol Π_{lottery} for aggregatable lotteries using a randomness beacon $\mathcal{F}_{\text{random}}$ (see full version [21]) and a broadcast channel $\mathcal{F}_{\text{broadcast}}$ (see full version [21]). Parties register for the lottery by running $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(\mathsf{par})$ and broadcasting their public key pk to other parties who verify it using VerKey. To commence the next lottery, the random beacon samples $\mathsf{lseed} \leftarrow_{\$} \{0,1\}^\lambda$ and distributes it to all the parties, each party then locally computes a ticket ticket as $\mathsf{ticket} \leftarrow \mathsf{Participate}(t, \mathsf{lseed}, \mathsf{pid}, \mathsf{sk})$ where t is the index of the current lottery, to observe whether they obtained a winning ticket for the current lottery. These tickets can be verified and aggregated by any party. Given a set of tickets $(\mathsf{ticket}_j)_{j=1}^L$ a party can locally use $\mathsf{Aggregate}$ to compute an aggregated ticket agg, similarly it can locally use Ver to verify an aggregated ticket agg. The full formal description and UC security proof can be found in our full version [21].

5 Discussion and Efficiency

In the final section of our paper, we present some practical thoughts about our construction and evaluate its concrete efficiency.

5.1 Practical Considerations

In practice, one can make some natural adjustments to our lottery, which we discuss here.

Weighted Lotteries. One may assign a weight p_j to participant j (e.g., based on its stake) such that j wins independently with probability p_j. We can adjust our lottery to support this: we simply let a participant with weight $p_j = 1/k_j$ commit to vectors over the range $[k_i]$ instead of $[k]$, and let the hash function defining j's challenge $x_j = \mathsf{H}(\mathsf{pk}_j, \mathsf{pid}_j, i, \mathsf{lseed})$ map to the range $[k_i]$.

Late Registration. Consider the case of ℓ lotteries and assume that a user registers late, say after the ith and before the $(i+1)$st lottery. In the extreme case, we have $i = \ell - 1$. As written, the user would have to sample a random vector of length ℓ and commit to it, while only the coordinates from $i+1$ to ℓ would be relevant. After the ℓth lottery, the entire system restarts and the user would have to generate a new key and register again. This is wasteful. Fortunately, there are ways to deal with this: (1) the user could implicitly set the first i coordinates to 0, which makes committing much more efficient (in evaluation form, see below). A similar solution applies when the user only wants to take part in any small subset of lotteries; (2) the user could keep its key for the next lifecycle of the lottery system until after the ith lottery, where for the i'th lottery ($i' \leq i$) in the next lifecycle, it would use the i'th coordinate.

High Entropy. Our proof only relies on the fact that the lottery seed output by the randomness beacon has high entropy. It is actually not necessary that it is uniformly random.

Evaluation Form. When KZG [30] is used as a vector commitment (as in our case), we should avoid explicitly computing the interpolating polynomial. Instead, we can compute the KZG commitments and openings more efficiently in the Lagrange basis. For that, we need to assume that the KZG setup contains elements $g_1^{\lambda_i(\alpha)}$, where λ_i is the ith Lagrange polynomial with respect to our evaluation domain. Note that this can be publicly pre-computed from a standard KZG setup. Optimal for efficiency is the case where we choose our evaluation domain to be the group of roots of unity and the polynomials we work with have degree d such that $d+1$ is a power of two, meaning that $\ell + 2$ has to be a power of two. In this case, we can benefit from several tricks to compute commitments and openings efficiently [19]. We emphasize that this changes the way we compute commitments and openings, but not their final value, meaning that this has no negative impact on security.

Pre-computing Openings. Note that for participants of a lottery, the most time-critical part is not key generation, but rather the time it takes to participate and compute winning tickets (algorithm Participate). In our scheme, a winning ticket is just a KZG opening proof within our evaluation domain. While computing a single proof takes linear time in ℓ (the number of lotteries), we can instead pre-compute all KZG opening proofs right after key generation and before lotteries take place. This can be done efficiently [20].

5.2 Efficiency Evaluation

With these considerations in mind, we evaluate the efficiency of our aggregatable lottery scheme Jackpot. We focus on communication/storage and computation costs.

Lottery Schemes. We have implemented the following lottery schemes in Rust using the arkworks[4] framework. VRF-BLS: The VRF-based lottery [17,26]

[4] http://arkworks.rs.

instantiated with BLS signatures [10] over curve BLS12-381 and SHA-256; more precisely, a party with secret key sk wins in lottery i with seed lseed if $\mathsf{H}(\sigma) < t$ for appropriate $t = t(k)$, where σ is a BLS signature of i and lseed and H is a hash function; σ is the winning ticket; To verify L tickets, we use BLS batch verification and L individual hash operations; Note that batch verification is only possible if all parties sign the same message, which is why can not include the party's identifier in the signed message. This also means that two parties with the same public key do not win independently, which has to be taken care of by other means. Jackpot: Our lottery scheme, using curve BLS12-381 to implement KZG; we follow the practical considerations discussed above; we have also implemented the technique from [20] to optionally pre-compute all openings. For all of our benchmarks, we assume $k = 512$ and $\mathsf{lseed} \in \{0,1\}^{256}$. Our code is available at

https://github.com/b-wagn/jackpot.

Bandwidth and Memory Consumption. Public keys for Jackpot have size $2 \cdot 48 + 2 \cdot 32 = 160$ Bytes, whereas they have size 48 Bytes for VRF-BLS. Now, assume that L winning tickets have to be stored or communicated. For VRF-BLS, this requires a storage of (ignoring public keys and identifiers of winning parties) $48 \cdot L$ Bytes, whereas for Jackpot each ticket has size $48 + 32 = 80$ Bytes, but the L tickets can be aggregated into one. This means that for L tickets, the relative improvement of Jackpot in comparison to VRF-BLS is $48 \cdot L/80 = 0.6 \cdot L$, which is a significant improvement even for small L. Table 1 shows some example numbers.

Table 1. Comparison of the memory consumption for storing or communicating L winning tickets for Jackpot with VRF-BLS. Memory is given in Bytes, and the column "Ratio" describes the ratio between the two schemes. We do not count player identifiers and their public keys, as they have to be stored on registration independent of winning tickets.

Tickets L	VRF-BLS [B]	Jackpot [B]	Ratio VRF-BLS/Jackpot
1	48	80	0.6
16	768	80	9.6
256	12288	80	153.6
1024	49152	80	614.4
2048	98304	80	1228.8

System Setup. For the following benchmarks, we have used a Macbook Pro (2020) with an Apple M1 processor, 16 GB of RAM, and MacOS Ventura 13.4. We benchmark our Rust implementation using the Criterion benchmark crate[5].

[5] https://github.com/bheisler/criterion.rs.

Aggregation and Verification. As our first benchmark in terms of running time, we evaluate the running times of aggregation (for Jackpot) and verification (for Jackpot and VRF-BLS) for different numbers L of tickets in Table 2. The running time is independent of the total number of lotteries. For VRF-BLS, we use BLS batch verification to verify multiple tickets with only one pairing equation. In this way, both schemes require L many hash evaluations and one pairing equation. Remarkably, our results demonstrate a increase in efficiency by a factor of 2 for Jackpot in comparison to VRF-BLS. This advantage arises from the fact that BLS batch verification necessitates operations over $\mathbb{G}_2$, whereas Jackpot's verification exclusively relies on operations over $\mathbb{G}_1$.

Key Generation and Pre-computation. In Table 3, we evaluate the parts of our scheme Jackpot for which the running time and memory depend on the total number of lotteries ℓ. For VRF-BLS, the running time is independent of ℓ and there is no preprocessing, and thus VRF-BLS is not listed in this table. We focus on three parameter settings $\ell \approx 2^z$ for $z \in \{10, 15, 20\}$. The table also shows the lifetime of keys for such settings, assuming one lottery every 5 minutes. In this case, 2^{20} lotteries are sufficient for 10 years. For these three parameter sets, we evaluate the running time of key generation (algorithm Gen) and the pre-computation of all KZG openings (algorithm Precompute). The table also shows the memory consumption for storing the result of the pre-computation.

Table 2. Benchmarked running times for aggregation and verification of L tickets for Jackpot and VRF-BLS. All times are given in milliseconds.

		$L = 1$	$L = 16$	$L = 256$	$L = 1024$	$L = 2048$
Jackpot	Aggregate [ms]	0.038	0.390	2.377	6.899	14.242
Jackpot	Ver [ms]	1.413	1.959	3.948	8.875	15.422
VRF-BLS	Ver [ms]	1.663	2.990	7.959	19.010	33.838

Table 3. Benchmark results of running times for our lottery scheme Jackpot for key generation (Gen), pre-computing all openings (Precompute), and participating (Participate and ComputeTicket) in a lottery for different numbers of lotteries $\ell = 2^z - 2$, $z \in \{10, 15, 20\}$. Lifetimes are estimated by assuming one lottery every 5 minutes.

Number of Lotteries	$\ell \approx 2^{10}$	$\ell \approx 2^{15}$	$\ell \approx 2^{20}$
Lifetime	3.5 days	4 months	10 years
Time Gen	14.83 ms	317.82 ms	8.27 s
Time Precompute	2.20 s	65.45 s	45 min
Memory Precompute	147 KB	5 MB	151 MB
Time Participate	1.26 μs	1.27 μs	1.31 μs
Time ComputeTicket	9.45 ms	215.80 ms	6.36 s

Functionality $\mathcal{F}_{\text{lottery}}(p, T)$

The functionality for T lotteries with winning probability p interacts with parties Player_j and a dedicated party $\mathsf{Player}_{\mathsf{Croupier}}$. It also interacts with the ideal world adversary Sim.

Interface Initialize:

```
$\mathsf{Tickets} := \emptyset$,   $\mathsf{AggTickets} := \emptyset$
$\mathsf{Win} := \emptyset$,   $\mathsf{Registered} := \emptyset$,   $\mathsf{LotCnt} := 1$
```

Interface Register: On $(\textsc{Register}, j, \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_{\mathsf{Croupier}}$

```
// Check if pid already registered
if $\exists(\_, \mathsf{pid}, \_) \in \mathsf{Registered}$ : return
$\mathsf{Sim} \overset{\mathsf{send}}{\hookleftarrow} (\textsc{Register}, j, \mathsf{pid})$

// Corrupted parties can win with smaller probability
if $\mathsf{Player}_j$ is corrupted:
    $\mathbf{p} \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_j$
    for $t \in [T]$ :  $\mathbf{p}_t := \min\{\mathbf{p}_t, p\}$
else :
    $\mathbf{p} := (p, \dots, p) \in \{p\}^T$

// Add to registrations
$\mathsf{Registered} := \mathsf{Registered} \cup \{(j, \mathsf{pid}, \mathbf{p})\}$
$\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\textsc{Register}, j, \mathsf{pid})$
```

Interface NextLottery: On $(\textsc{NextLottery}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_{\mathsf{Croupier}}$

```
// Obtain a label for this lottery
if $\mathsf{LotCnt} > T$ : return
$l \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Sim}$

// Sample lottery outputs for all registered parties
for $(j, \mathsf{pid}, \mathbf{p}) \in \mathsf{Registered}$ :
    $w \leftarrow \mathcal{B}(\mathbf{p}_{\mathsf{LotCnt}})$    // Sample from Bernoulli distribution
    if $w = 1$ : $\mathsf{Win} := \mathsf{Win} \cup \{(l, j, \mathsf{pid})\}$
    $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} (\textsc{NextLottery}, \mathsf{pid}, l, w)$

// Advance lottery counter
$\mathsf{LotCnt} := \mathsf{LotCnt} + 1$
```

Fig. 5. Ideal functionality $\mathcal{F}_{\text{lottery}}(p, T)$ for an T-time aggregatable lottery with winning probability p. The remaining interfaces are given in Fig. 6.

Even for 2^{20} lotteries, running times and memory consumption remain within practical bounds. Especially, the pre-computation's duration of approximately 45 minutes is acceptable, given that it can be run in the background at the user's convenience, and it is a one-time task for the entire lifespan of a key.

```
Functionality $\mathcal{F}_{\text{lottery}}(p, T)$ (continued)

Interface PARTICIPATE: On $(\textsc{Participate}, l, \mathsf{pid}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_j$
  // Obtain fresh ticket label from Sim
01 $\mathsf{Sim} \overset{\mathsf{send}}{\hookleftarrow} (\textsc{Participate}, l, \mathsf{pid})$
02 $\mathsf{ticket} \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Sim}$

  // If player won add ticket to table
03 if $(l, \mathsf{pid}, j) \in \mathsf{Win}$ :
04    $\mathsf{Tickets} := \mathsf{Tickets} \cup \{(l, \mathsf{pid}, \mathsf{ticket})\}$
05    $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \mathsf{ticket}$
06 else $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \bot$

Interface AGGREGATE:
On $(\textsc{Aggregate}, l, I = (\mathsf{pid}_{j'})_{j' \in [L]}, T = (\mathsf{ticket}_{j'})_{j' \in [L]}\}) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_j$
  // Check that the tickets are winning
01 for $(\mathsf{pid}, \mathsf{ticket}) \in T$ :
02    if $(l, \mathsf{pid}, \mathsf{ticket}) \notin \mathsf{Tickets}$ :
03       $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \bot$
04       return

  // Obtain aggregated ticket label from Sim
05 $\mathsf{Sim} \overset{\mathsf{send}}{\hookleftarrow} (\textsc{Aggregate}, l, I, T)$
06 $\mathsf{agg} \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Sim}$

  // Store aggregated ticket label
07 $\mathsf{AggTickets} := \mathsf{AggTickets} \cup \{(l, \{\mathsf{pid} : \mathsf{pid} \in I\}, \mathsf{agg})\}$
08 $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \mathsf{agg}$

Interface VERIFY: $(\textsc{Verify}, l, \mathsf{agg}, I) \overset{\mathsf{recv}}{\hookleftarrow} \mathsf{Player}_j$
  // Only tickets generated by the functionality are valid
01 $\mathsf{Sim} \overset{\mathsf{send}}{\hookleftarrow} (\textsc{Verify}, l, \mathsf{agg}, I)$
02 if $(l, I, \mathsf{agg}) \in \mathsf{AggTickets}$ :
03    $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \top$
04 else
05    $\mathsf{Player}_j \overset{\mathsf{send}}{\hookleftarrow} \bot$
```

Fig. 6. Remaining interfaces of ideal functionality $\mathcal{F}_{\text{lottery}}(p, T)$ for T aggregatable lotteries with winning probability p. The other interfaces are given in Fig. 5.

Participation. To participate in a lottery round, a party determines whether it won, and it computes a winning ticket in case it did. While we combined these two tasks in our modeling (algorithm Participate), we separate them for our benchmarks (algorithms Participate and ComputeTicket, respectively). We show the results in Table 3. For all parameter sets, the running time of Participate (i.e., checking if the party won) is negligibly small, namely, within microseconds. The running time to compute a ticket scales linearly with ℓ, but even for 2^{20} lotteries it is within a practical range: waiting 7 seconds for a ticket is fine, as one can compute a ticket in our scheme even before the lottery round started. Also, assuming we did a pre-computation as discussed above, this cost is completely avoided.

References

1. Bartoletti, M., Zunino, R.: Constant-deposit multiparty lotteries on bitcoin. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) FC 2017 Workshops. LNCS, vol. 10323, pp. 231–247. Springer, Heidelberg (Apr 2017)
2. Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 263–280. Springer, Heidelberg (Apr 2012). https://doi.org/10.1007/978-3-642-29011-4_17
3. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Denning, D.E., Pyle, R., Ganesan, R., Sandhu, R.S., Ashby, V. (eds.) ACM CCS 93. pp. 62–73. ACM Press (Nov 1993). https://doi.org/10.1145/168588.168596
4. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part II. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (Aug 2014). https://doi.org/10.1007/978-3-662-44381-1_24
5. Boneh, D., Boyen, X.: Short signatures without random oracles and the SDH assumption in bilinear groups. Journal of Cryptology **21**(2), 149–177 (Apr 2008). https://doi.org/10.1007/s00145-007-9005-7
6. Boneh, D., Drijvers, M., Neven, G.: Compact multi-signatures for smaller blockchains. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018, Part II. LNCS, vol. 11273, pp. 435–464. Springer, Heidelberg (Dec 2018). https://doi.org/10.1007/978-3-030-03329-3_15
7. Boneh, D., Eskandarian, S., Hanzlik, L., Greco, N.: Single secret leader election. In: AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020. pp. 12–24. ACM (2020), https://doi.org/10.1145/3419614.3423258
8. Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P.M.R., Sahai, A.: Threshold cryptosystems from threshold fully homomorphic encryption. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 565–596. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_19
9. Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 416–432. Springer, Heidelberg (May 2003). https://doi.org/10.1007/3-540-39200-9_26
10. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the Weil pairing. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 514–532. Springer, Heidelberg (Dec 2001). https://doi.org/10.1007/3-540-45682-1_30
11. Broder, A.Z., Dolev, D.: Flipping coins in many pockets (byzantine agreement on uniformly random values). In: 25th FOCS. pp. 157–170. IEEE Computer Society Press (Oct 1984). https://doi.org/10.1109/SFCS.1984.715912

12. Camenisch, J., Drijvers, M., Gagliardoni, T., Lehmann, A., Neven, G.: The wonderful world of global random oracles. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part I. LNCS, vol. 10820, pp. 280–312. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78381-9_11
13. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press (Oct 2001). https://doi.org/10.1109/SFCS.2001.959888
14. Catalano, D., Fiore, D.: Vector commitments and their applications. In: Kurosawa, K., Hanaoka, G. (eds.) PKC 2013. LNCS, vol. 7778, pp. 55–72. Springer, Heidelberg (Feb / Mar 2013). https://doi.org/10.1007/978-3-642-36362-7_5
15. Chen, B., Bünz, B., Boneh, D., Zhang, Z.: HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023, Part II. LNCS, vol. 14005, pp. 499–530. Springer, Heidelberg (Apr 2023). https://doi.org/10.1007/978-3-031-30617-4_17
16. Chow, S.S., Hui, L.C., Yiu, S.M., Chow, K.: An e-lottery scheme using verifiable random function. In: International Conference on Computational Science and its Applications. pp. 651–660. Springer (2005)
17. David, B., Gazi, P., Kiayias, A., Russell, A.: Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 66–98. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78375-8_3
18. Faonio, A., Fiore, D., Kohlweiss, M., Russo, L., Zajac, M.: From polynomial IOP and commitments to non-malleable zksnarks. IACR Cryptol. ePrint Arch. p. 569 (2023), https://eprint.iacr.org/2023/569
19. Feist, D.: PCS multiproofs using random evaluation. https://dankradfeist.de/ethereum/2021/06/18/pcs-multiproofs.html (2021), accessed: 2023-09-28
20. Feist, D., Khovratovich, D.: Fast amortized KZG proofs. Cryptology ePrint Archive, Report 2023/033 (2023), https://eprint.iacr.org/2023/033
21. Fleischhacker, N., Hall-Andersen, M., Simkin, M., Wagner, B.: Jackpot: Non-interactive aggregatable lotteries. Cryptology ePrint Archive, Paper 2023/1570 (2023), https://eprint.iacr.org/2023/1570
22. Fleischhacker, N., Herold, G., Simkin, M., Zhang, Z.: Chipmunk: Better synchronized multi-signatures from lattices. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022. pp. 1109–1123. ACM (2022), https://doi.org/10.1145/3548606.3560655
23. Fleischhacker, N., Simkin, M., Zhang, Z.: Squirrel: Efficient synchronized multi-signatures from lattices. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022. pp. 1109–1123. ACM Press (Nov 2022). https://doi.org/10.1145/3548606.3560655
24. Fuchsbauer, G., Kiltz, E., Loss, J.: The algebraic group model and its applications. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part II. LNCS, vol. 10992, pp. 33–62. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96881-0_2
25. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: 54th FOCS. pp. 40–49. IEEE Computer Society Press (Oct 2013). https://doi.org/10.1109/FOCS.2013.13
26. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: Proceedings of the 26th Symposium

on Operating Systems Principles, Shanghai, China, October 28-31, 2017. pp. 51–68. ACM (2017), https://doi.org/10.1145/3132747.3132757
27. Gorbunov, S., Reyzin, L., Wee, H., Zhang, Z.: Pointproofs: Aggregating proofs for multiple vector commitments. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 2007–2023. ACM Press (Nov 2020). https://doi.org/10.1145/3372297.3417244
28. Groth, J., Maller, M.: Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part II. LNCS, vol. 10402, pp. 581–612. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63715-0_20
29. Itakura, K.: A public-key cryptosystem suitable for digital multisignature. NEC research and development **71**, 1–8 (1983)
30. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 177–194. Springer, Heidelberg (Dec 2010). https://doi.org/10.1007/978-3-642-17373-8_11
31. Liang, B., Banegas, G., Mitrokotsa, A.: Statically aggregate verifiable random functions and application to e-lottery. Cryptography **4**(4), 37 (2020)
32. Libert, B., Ling, S., Nguyen, K., Wang, H.: Zero-knowledge arguments for lattice-based accumulators: Logarithmic-size ring signatures and group signatures without trapdoors. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part II. LNCS, vol. 9666, pp. 1–31. Springer, Heidelberg (May 2016). https://doi.org/10.1007/978-3-662-49896-5_1
33. Libert, B., Passelègue, A., Riahinia, M.: PointProofs, revisited. In: Agrawal, S., Lin, D. (eds.) ASIACRYPT 2022, Part IV. LNCS, vol. 13794, pp. 220–246. Springer, Heidelberg (Dec 2022). https://doi.org/10.1007/978-3-031-22972-5_8
34. Libert, B., Yung, M.: Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In: Micciancio, D. (ed.) TCC 2010. LNCS, vol. 5978, pp. 499–517. Springer, Heidelberg (Feb 2010). https://doi.org/10.1007/978-3-642-11799-2_30
35. Libert, B.: Simulation-extractable KZG polynomial commitments and applications to HyperPlonk. In: PKC 2024. LNCS, Springer, Heidelberg (May 2024), to appear
36. Micali, S., Ohta, K., Reyzin, L.: Accountable-subgroup multisignatures: Extended abstract. In: Reiter, M.K., Samarati, P. (eds.) ACM CCS 2001. pp. 245–254. ACM Press (Nov 2001). https://doi.org/10.1145/501983.502017
37. Micali, S., Rabin, M.O., Vadhan, S.P.: Verifiable random functions. In: 40th FOCS. pp. 120–130. IEEE Computer Society Press (Oct 1999). https://doi.org/10.1109/SFFCS.1999.814584
38. Papamanthou, C., Shi, E., Tamassia, R., Yi, K.: Streaming authenticated data structures. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 353–370. Springer, Heidelberg (May 2013). https://doi.org/10.1007/978-3-642-38348-9_22
39. Sahai, A.: Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In: 40th FOCS. pp. 543–553. IEEE Computer Society Press (Oct 1999). https://doi.org/10.1109/SFFCS.1999.814628
40. Schnorr, C.P.: Efficient signature generation by smart cards. Journal of Cryptology **4**(3), 161–174 (Jan 1991). https://doi.org/10.1007/BF00196725
41. Tomescu, A., Abraham, I., Buterin, V., Drake, J., Feist, D., Khovratovich, D.: Aggregatable subvector commitments for stateless cryptocurrencies. In: Galdi, C., Kolesnikov, V. (eds.) SCN 20. LNCS, vol. 12238, pp. 45–64. Springer, Heidelberg (Sep 2020). https://doi.org/10.1007/978-3-030-57990-6_3

Early Stopping Byzantine Agreement in $(1 + \epsilon) \cdot f$ Rounds

Fatima Elsheimy[1](✉), Julian Loss[2], and Charalampos Papamanthou[1]

[1] Yale University, Connecticut, USA
`Fatima.Elsheimy@yale.edu`

[2] CISPA Helmholtz Center for Information Security, Saarbücken, Germany

Abstract. In this paper, we present two *early stopping* Byzantine agreement protocols in the authenticated setting against a corrupt minority $t < n/2$, where t represents the maximum number of malicious parties. Early stopping protocols ensure termination within a number of rounds determined solely by the actual number of malicious nodes f present during execution, irrespective of t.

Our first protocol is deterministic and ensures early stopping termination in $(d + 5) \cdot (\lfloor f/d \rfloor + 2) + 2$ rounds, where d is a fixed constant. For example, for all $d \geq 6$, our protocol runs in at most $(1 + \epsilon) \cdot f$ rounds (where $0 < \epsilon < 1$), improving (for large f) upon the best previous early stopping deterministic broadcast protocol by Perry and Toueg [21], which terminates in $min(2f + 4, 2t + 2)$ rounds. Additionally, our second protocol is randomized, ensuring termination in an expected constant number of rounds and achieving early stopping in $(d + 9) \cdot (\lfloor f/d \rfloor + 1) + 2$ rounds in the worst case. This marks a significant improvement over a similar result by Goldreich and Petrank. [15], which *always* requires an expected constant number of rounds and $O(t)$ rounds in the worst case, i.e., does not have the early stopping property.

1 Introduction

Byzantine Agreement (BA) is a fundamental problem in distributed computing. In the BA problem, n parties start with some value in $\{0, 1\}$ and wish to jointly agree on one value while tolerating up to $t < n/2$ Byzantine parties (Agreement.) If all honest parties start with the same value, they must output that value (Validity.) The foundations of this field were established by the pioneering work of Lamport, Shostak, and Pease in the 1980s [17]. One of the main metrics of efficiency for BA protocols is their *round complexity*, i.e., the number of synchronous interactions required for the protocol to terminate. This is the focus of our paper.

This holds for all $d \geq 6$ and $f > \frac{2d^2+8d}{d-5}$, where d is a predefined fixed constant. In general, our protocol achieves a round complexity of $(1 + O(1/d)) \cdot f + O(d)$, which simplifies to $(1 + \epsilon) \cdot f$ whenever d behaves as a constant in f.

K.-M. Chung and Y. Sasaki (Eds.): ASIACRYPT 2024, LNCS 15489, pp. 398–424, 2025.
https://doi.org/10.1007/978-981-96-0938-3_13

A seminal result by Dolev and Strong [9][1] demonstrates that any BA protocol capable of tolerating $t < n/2$ malicious parties necessitates at least $t + 1$ rounds in some runs. However, this bound is considered loose for protocol executions where the number of corruptions, f, is less than t. According to Dolev et al. [8], the round complexity lower bound in this scenario is $\min\{f + 2, t + 1\}$. Thus, a series of works studies *early stopping protocols* that terminate based solely on the actual number of corruptions f. For the information-theoretic setting and $t < n/3$, this has culminated in the work of Abraham and Dolev [2] who gave the first early stopping protocol with polynomial communication and optimal round complexity of $\min\{f + 2, t + 1\}$. By comparison, the authenticated setting (where signatures can be used) with $t < n/2$ malicious corruptions is far less explored. To the best of our knowledge, the only early stopping protocol in this setting is due to Perry and Toueg [21] which has (sub-optimal) round complexity $\min\{2f + 4, 2t + 2\}$. This raises the following natural question: *Is there an early-stopping protocol for authenticated Byzantine agreement with* $t < n/2$ *corruptions which approaches the lower bound of* $\min\{f + 2, t + 1\}$? We answer this question affirmatively by showing the following results:

– We begin by proving a deterministic early-stopping Byzantine agreement protocol that terminates in $(d + 5) \cdot (\lfloor f/d \rfloor + 2) + 2$ rounds, where d is a fixed positive constant. In particular, for all $d \geq 6$ and
$$f > \frac{2d^2 + 8d}{d - 5}$$
our protocol always outperforms Perry and Toueg's protocol. In general, our protocol achieves a round complexity of
$$(1 + O(1/d)) \cdot f + O(d)\,,$$
which simplifies to $(1 + \epsilon) \cdot f$ whenever d behaves as a constant in f.
– We then show an early stopping randomized Byzantine agreement protocol with expected constant rounds, whose worst-case round complexity is $(d+9) \cdot (\lfloor f/d \rfloor + 1) + 2$, where again, d is a predefined constant. Our protocol compares favorably with protocols obtained via the generic compiler of Goldreich and Petrank [15]. Like our work, their compiler gives an expected constant round protocol, but its worst-case round complexity is $O(t)$—therefore it does not yield an early stopping protocol.

At the heart of our construction, we devise a novel method of eliminating faulty parties that keep the protocol from terminating. Our construction relies on prior work of Fitzi and Nielsen [11] to improve the ratio of eliminated parties to protocol rounds. On average, our protocol eliminates 1 party every $1 + 5/d$ rounds, whereas the protocol of Perry and Toueg's protocol eliminates 1 party every 2 rounds. We now explain our techniques in more detail.

[1] [9] presents the result for Byzantine Broadcast, a variant of Byzantine Agreement in which a designated sender sends an input value to other parties who must reach consensus on this value. There is a known reduction to Byzantine Agreement with optimal resilience of $t < n/2$.

1.1 Technical Overview

Correct-or-Detect Broadcast. We begin by recalling the Correct-Or-Detect Broadcast protocol of Fitzi and Nielsen which forms the basis of our construction. Their protocol, henceforth denoted $\Pi_{\mathsf{d\text{-}CoD}}$ [11], is parametrized by an arbitrary positive integer d and a designated sender P_s and runs in $d+4$ rounds. $\Pi_{\mathsf{d\text{-}CoD}}$ is based on the seminal broadcast protocol of Dolev and Strong, which itself runs in $t+1$ rounds and is secure against any number of $t < n$ corrupted parties. However, $\Pi_{\mathsf{d\text{-}CoD}}$ is a binary protocol, meaning the sender can have a value of either 0 or 1. It is also 1-biased: the designated sender sends their value to all parties if the value is 1, but refrains from sending anything otherwise. Rather than achieving full broadcast, parties in $\Pi_{\mathsf{d\text{-}CoD}}$ terminate the protocol in two possible modes C (correct) and D (detect). In case an honest party terminates in mode C, $\Pi_{\mathsf{d\text{-}CoD}}$ achieves the properties of broadcast, i.e., all parties agree on the sender's value. Moreover, if the sender P_s is honest, all honest parties always terminate in mode C. On the other hand, if some honest party terminates in mode D, $\Pi_{\mathsf{d\text{-}CoD}}$ may not achieve the properties of broadcast. Yet, in this case, the protocol ensures that all parties identify a common set of d corrupted parties. To this end, every party P_i among the set of honest parties $\mathcal{H}$ outputs a list $\mathcal{F}_i$ of parties it knows to be corrupted, where the protocol ensures that $|\bigcap_{P_i \in \mathcal{H}} \mathcal{F}_i| \geq d$. It is important to note that there is no agreement among parties on what mode the protocol terminates in (otherwise, $\Pi_{\mathsf{d\text{-}CoD}}$ would be a full-fledged broadcast protocol). We extend the construction of Fitzi and Nielsen for binary messages to messages of arbitrary length in the straight-forward way by broadcasting a message bit by bit and determining the termination mode as C iff all of the bit-wise sub-instances output C. Otherwise, we output D and take the union of identified malicious sets output in any of these instances.

As we use $\Pi_{\mathsf{d\text{-}CoD}}$ as a subroutine, it is crucial to ensure that the agreed-upon d faulty parties cannot participate in future invocations of $\Pi_{\mathsf{d\text{-}CoD}}$. Therefore, at the beginning of $\Pi_{\mathsf{d\text{-}CoD}}$, parties issue each other *proofs of participation* (PoP). Specifically, party P_i sends a signature to party P_j if P_j is not in the $\mathcal{F}_i$. Since honest parties are never included in each other's faulty lists, each honest party receives a PoP, allowing them to continue participating in the protocol. On the other hand, parties identified as corrupt do not receive a PoP and are thereby excluded from the protocol. To enforce this, each party attaches its PoP to every message it sends within the protocol. Additionally, parties will only accept messages from party P_j if they are accompanied by P_j's PoP. This method ensures that each new invocation of $\Pi_{\mathsf{d\text{-}CoD}}$ successfully identifies and excludes d new malicious parties.

Graded Consensus with Detection. We now explain our main technical building block, which we refer to as *graded consensus with detection*. For simplicity, we focus here on our basic version of this primitive in which all parties input a binary value v_i along with their current list $\mathcal{F}_i$ of faulty parties. We additionally require that honest parties are never in each others list of identified corrupted parties.

The protocol outputs a value $y_i \in \{0,1\}$ along with a grade $g_i \in \{0,1\}$ and an updated list $\mathcal{F}_i^*$ of faulty parties. As with existing constructions of graded consensus in the literature, our protocol uses the grade g_i to indicate a party's confidence in its output y_i. *Graded consistency* says that on outputting grade $g_i = 1$, P_i knows that all parties agree on P_i's output y_i, but they might not know that they agree (as they have output grade 0). On the other hand, we ensure *graded validity*: if all honest parties input the same value v to the protocol, then all honest parties output $y_i = v$ and grade $g_i = 1$.

The distinguishing feature of our new construction is to ensure that if two honest parties P_i and P_j disagree on their respective outputs $y_i \neq y_j$, then they identify a common set of at least d corrupted parties and extend their faulty lists $\mathcal{F}_i^*$ accordingly. Importantly, we can ensure that the intersection $\bigcap_{P_i \in \mathcal{H}} \mathcal{F}_i^*$ contains at least d corrupt parties that are not contained in the common set of parties' faulty input lists $\bigcap_{P_i \in \mathcal{H}} \mathcal{F}_i$. Because the faulty lists of honest parties can never contain honest parties, this automatically implies that parties agree on their output (albeit possibly with grade 0) once there are fewer than d malicious parties. This property will be crucially exploited in our overall construction of Byzantine Agreement.

From CoD-Broadcast to Graded Consensus with Detection. Our construction is remarkably simple and builds on the multivalued CoD-Broadcast described earlier. Each party sends its input (i, v_i) via $\Pi_{\mathsf{d\text{-}CoD}}$ to all other parties. Including the identifier i with v_i is crucial because v_i might be 0, and without i, P_i would not send its value to any party according to $\Pi_{\mathsf{d\text{-}CoD}}$. Honest parties would then be unable to distinguish between an honest party with a value of 0 and a malicious party that does not have a valid PoP and therefore cannot send anything. By ensuring that each message includes a non-zero component, we guarantee that all parties send a non-zero message, allowing honest parties to thereby confirm the honesty of the sender. To determine the output, we let parties take a majority over all the instances that were received with output $s \neq \perp$, which means those instances belong to parties with valid PoP from $\Pi_{\mathsf{d\text{-}CoD}}$. To output $y_i = v$ with grade $g_i = 1$, a party P_i waits to observe $t+1$ instances terminate on value v in mode C (and with output $s \neq \perp$, where s is the identifier of the sender P_s). On the other hand, for grade $g_i = 0$, P_i simply takes the majority bit over all instances with $s \neq \perp$ (regardless of what mode they terminate in). From the properties of $\Pi_{\mathsf{d\text{-}CoD}}$, it immediately follows that the usual consistency and validity properties of graded consensus. On the other hand, disagreement can only happen if at least one of the $\Pi_{\mathsf{d\text{-}CoD}}$ instances terminates in mode D. In this case, all parties can update their lists $\mathcal{F}_i^*$ with a common set of at least d newly identified malicious parties. Moreover, our protocol adds only 1 round (for PoPs) to the running time of $\Pi_{\mathsf{d\text{-}CoD}}$, thus coming out to a total running time of $d+5$ rounds.

From GC with Detection to Deterministic Byzantine Agreement. We run the detecting graded consensus protocol described above in iterations. In each iteration k, parties update their input $v_i, \mathcal{F}_i$ to the output value y_i and faulty list $\mathcal{F}_i^*$ of iteration $k-1$. A party P_i terminates after observing the graded consensus protocol outputting grade $g_i = 1$ in some iteration k and running for one more subsequent iteration. By graded validity, this ensures that parties all parties observe the same condition by iteration $k + 1$ and can terminate by iteration $k + 2$ at the latest. The detection property of our graded consensus module ensures that in every iteration where parties do not terminate, they all add d common parties to their list of identified corrupted parties. If there are less than d malicious parties left, honest parties still output the same value. Thus, after at most $\lfloor f/d \rfloor$ iterations, all remaining parties must output the same value. By the above argument, this ensures that they terminate within at most three more iterations; one iteration to output the same value and two more from the above argument. Since each iteration takes $d+5$ rounds, our running time comes out to $(d + 5) \cdot (\lfloor f/d \rfloor + 3)$ many rounds.

Randomized Early Stopping Agreement. We conclude by explaining how to randomize the protocol sketched above. In this manner, we obtain an expected constant round protocol which also has early stopping complexity $(d+9)\cdot(\lfloor f/d \rfloor+2)$. To this end, we add a few rounds on top of our detecting graded consensus protocol so as to obtain a stronger version of graded consensus with three possible grades $0, 1,$ and 2. Here grade 2 indicates the highest confidence in a binary output y_i and indicates agreement for any party who observes it. Thus, a party P_i sets terminates after observing the graded consensus protocol outputting grade $g_i = 2$ and running for one more subsequent iteration. On the other hand, grade 1 leaves open the possibility that another honest party has grade 0, in which case its corresponding output is the default value $\perp$. Our construction also extends the properties of the detecting properties of the $(0, 1)$ graded consensus protocol described above in the natural way and ensures that once no corrupted parties remain, parties always agree on their output.

Using this strengthened version of detecting graded consensus, we are able to run a standard construction of randomized byzantine agreement from graded consensus. As before, we iterate instances of graded consensus and input the output from the current iteration to the next iteration. However, parties update their input to the next iteration to a common random coin whenever it outputs $\perp$ with grade 0 in some iteration of the protocol. If the coin agrees for all parties with some constant probability p, this ensures that parties agree on what they input to any iteration with probability at least $p/2$. Thus, parties terminate the protocol in $O(2/p) = O(1)$ expected iterations of constant round length. The exact round complexity in expectation is $((2/p) + 2)(d + 9)$, where $d + 9$ are the number of rounds in an iteration. On the other hand, we can argue along the same lines as for the deterministic case that all parties terminate in the worst case after $\lfloor f/d \rfloor + 2$ iterations, i.e., there are less than d dishonest parties left to obstruct termination.

Optimization. In our current Byzantine agreement protocols, each party runs an additional iteration after setting its output to assist other parties in setting their outputs and ensuring agreement. However, there are scenarios where all honest parties may set their output in the same iteration, resulting in an unnecessary extra iteration. To circumvent this inefficiency, we rely termination certificates. Once an honest party sets its output $y_i = \mathsf{v}$, it sends a termination certificate $\langle \mathsf{terminate}, \mathsf{v}\rangle_i$ to all parties. From the agreement property of the protocols, all honest parties will send termination certificates for the same value. Thus, if a party receives t+1 termination certificates for the same value v, it sets its output value if it wasn't set before, forwards the t+1 certificates in the next round, and then terminates. As discussed, this could potentially save an unnecessary iteration. Consequently, the round complexity improves to $(d+5)\cdot(\lfloor f/d\rfloor+2)+2$ and $(d+9)\cdot(\lfloor f/d\rfloor+1)+2$ for determinstic BA and randomized BA protocols, respectively.

1.2 Related Work

Byzantine agreement has been extensively studied since the pioneering work of Shostak, Pease, and Lamport [17]. Dolev and Strong [9] established a critical result, showing that any broadcast protocol tolerating $t < n$ malicious parties requires at least $t+1$ rounds. However, this bound was later refined by Dolev et al. [8], who demonstrated that when the number of corruptions, f, is much less than t, the lower bound is $\min(f+2, t+1)$. Since then, significant progress has been made in developing early stopping protocols.

The first such protocol in the information-theoretic setting with optimal resilience $t < n/3$ was introduced by Berman et al. [4], though it suffered from exponential communication complexity. Garay and Moses later addressed this issue, presenting a Byzantine agreement protocol with polynomial-sized messages but slightly suboptimal early stopping round complexity of $\min(f+5, t+1)$ [13,14]. More recently, Abraham and Dolev [2] achieved a breakthrough by developing the first early stopping protocol with polynomial communication, optimal resilience, and optimal round complexity of $\min(f+2, t+1)$. While the information-theoretic setting has seen extensive research, there has been limited work in the authenticated setting with optimal resilience $t < n/2$. To the best of our knowledge, Perry and Toueg [21] provide the only authenticated early stopping protocol with polynomial communication and a round complexity of $\min(2f+4, 2t+2)$.

As for randomized protocols, it has been established that they can achieve an expected constant number of rounds in both the information-theoretic setting [10] and the authenticated setting [1,16,22]. However, these protocols have a negligible probability of very long runs due to their failure probability. Goldreich et al. [15] presented a method to eliminate the failure probability, achieving an expected constant round complexity and worst-case round complexity of $O(t)$ for up to $t < n/2$ corruptions—therefore it does not yield an early stopping protocol. A follow-up work further improved this, achieving expected constant round complexity and optimal worst-case complexity of $t+1$ rounds for a worse resilience of $t < n/8$ [23]. Achieving expected constant round complexity, $t+1$ rounds worst case, and optimal resilience $t < n/3$ remains unresolved. Importantly, this question remains open even without considering the early stopping worst-case round complexity. We note that it is possible to terminate randomized protocols in round complexity that is independent of the number of corrupted parties. However, in this case, the number of rounds always depends on the desired error probability δ of the protocol. This makes such protocols difficult to compare to early stopping protocols. In particular, early stopping protocols may require much fewer rounds to terminate when the number f of corruptions is low.

Other works [3,7,20] have explored early stopping protocols but in much weaker adversary settings, such as omission and crash adversary models. A recent of work of Loss and Nielsen [18] gives the first early stopping protocol for the dishonest majority setting with $t < n$ corruptions, albeit with significantly worse round complexity $O(\mathsf{min}\{f^2, t\})$.

1.3 Paper Organization

Section 2 provides definitions for Byzantine Agreement, $(0, 1)$-Graded, and $(0, 1, 2)$-Graded d-Detecting Byzantine Agreement, as well as for the cryptographic primitives we use, such as signature schemes and common coin. In Sect. 3, we discuss the intuition and construction of the deterministic Byzantine agreement protocol, along with its correctness proof. In Sect. 4, we present the intuition and construction of the randomized Byzantine agreement protocol. Finally, we propose a way to further optimize the round complexity in Appendix A. Some other supplementary protocols are also deferred to the Appendix.

2 Preliminaries

We begin by introducing the model as well as basic definitions.

Network and Setup Assumptions. We assume a fully connected network of pairwise, authenticated channels between n parties $\{P_1, ..., P_n\} = \mathcal{P}$. We consider the *synchronous network model* where all parties have access to a synchronized clock and there is a known upper bound Δ on the message delays of honest parties. This allows parties to run protocols in a round-by-round fashion where rounds are of length Δ and any message that is sent by an honest party at the beginning of a round are delivered by the end of that round to all honest parties. Parties are assumed to have established a public key infrastructure (PKI) of a digital signature scheme that provides an efficient signing routine Sign and an efficient verification routine Verify. Every party P_i is associated with a public key pk_i that is known to all parties and where (only) P_i knows the corresponding secret key sk_i. This allows a party P_i to create a signature $\langle m \rangle_i$ on message m using its secret key sk_i via $\langle m \rangle_i := \mathsf{Sign}(\mathsf{sk}_i, m)$. $\langle m \rangle_i$ can then be efficiently verified by running $\mathsf{Verify}(\mathsf{pk}_i, \langle m \rangle_i, m)$. We refer to a signature $\langle m \rangle_i$ as *valid* if $\mathsf{Verify}(\mathsf{pk}_i, \langle m \rangle_i, m) = 1$. For ease of notation, we use the abbreviated notation $\langle m \rangle_i$ to refer to tuples $(m, \mathsf{sign}(m, \mathsf{sk}_i))$ throughout the paper.

As discussed in the introduction, each protocol is designed to invoke other subroutines. Therefore, we implicitly assume that every protocol is associated with a session identifier ssid, which indicates the session in which the protocol is invoked. Consequently, if a proof certificate is created using messages exchanged during a session ssid, it will not be valid or applicable for use in any other session $\mathsf{ssid}' \neq \mathsf{ssid}$. To maintain clarity in our notation, we refrain from explicitly including ssid in our protocols.

Adversary Model. We consider an adaptive Byzantine adversary that can corrupt up to $t < n/2$ parties at any point of a protocol execution. In our protocols, the variable t is defined as $t = \lceil \frac{n-1}{2} \rceil$. We refer to the *actual number* of corruptions during an execution of the protocol as $f \leq t$. A corrupt (or malicious) party P_i is under full control of the adversary and may deviate arbitrarily from the protocol. In particular, the adversary learns P_i's signing key sk_i, which allows it to sign messages on P_i's behalf. In addition, we allow the adversary to delete (or replace with its own) any undelivered messages of a newly corrupted party P_i that P_i sent while it was still honest. We denote the set of uncorrupted (or honest) parties as $\mathcal{H}$.

We assume that the adversary is computationally bounded and cannot forge signatures of honest parties. In line with the literature in this area, we treat signatures as idealized primitives with perfect security. When instantiating the signature scheme with an existentially unforgeable one, we obtain protocols with non-neglible probability of failure.

Common Coin. We assume an ideal coin-flip protocol $\mathsf{CoinFlip}$ that allows parties to agree with constant probability $p < 1$ on a random coin in $\{0, 1\}$. This protocol can be viewed as an ideal functionality [6] that upon receiving input r from $t+1$ parties generates a random coin c_i and sends $(c_i^{(r)})$ to each party $P_i \in \mathcal{P}$, where $c_i^{(r)} = c_j^{(r)}$ with probability at least p. The value remains uniform from the adversary's view until the first honest party has queried $\mathsf{CoinFlip}$. Such a primitive can be achieved using verfiable random functions [19], threshold signatures [5], or verifiable secret sharing [16].

Next, we present definitions of well-known primitives, such as Byzantine agreement and graded consensus. Then, we introduce new definitions for our proposed protocols: graded consensus with detection.

Definition 1 (Byzantine Agreement). *Let Π be protocol executed among parties $P_1, ..., P_n$, where each party P_i holds an input $v_i \in \{0, 1\}$ and outputs a value $y_i \in \{0, 1\}$ upon terminating. A protocol Π achieves* Byzantine Agreement*, if the following properties hold whenever at most t parties are corrupted.*

- *Validity: If every honest party P_i inputs $v_i = \mathsf{v}$, then all honest parties output $y_i = \mathsf{v}$;*
- *Consistency: All honest parties output the same value v.*
- *Termination: Every honest party terminates.*

Definition 2 ($(0,1,2)$-Graded Agreement). *Let Π be a protocol executed by parties $P_1, ..., P_n$, where each party P_i inputs $v_i \in \{0, 1\}$ and outputs a value $y_i \in \{0, 1, \perp\}$ and a grade $g_i \in \{0, 1, 2\}$ upon terminating. A protocol Π achieves* $(0,1,2)$-Graded Agreement *if the following properties hold whenever at most t parties are corrupted.*

– *Graded Validity: If all honest parties P_i have the same input value $v_i = \mathsf{v}$ then all honest parties output $y_i = \mathsf{v}$ and $g_i = 2$*
– *Graded Consistency: Let P_i and P_j denote honest parties that output y_i, g_i and y_j, g_j, respectively. Then (1) $|g_i - g_j| \leq 1$ and (2) $g_i, g_j \geq 1$ implies that $y_i = y_j$*
– *Termination: Every honest party terminates.*

Next, we define the Correct or Detect Broadcast primitive. It is important to note that our definition differs from the one in [11]. In our version, each honest party P_i inputs a faulty list $\mathcal{F}_i$ along with its initial value v_i. Essentially, malicious parties included in the initial faulty list of all honest parties are excluded from participating in the protocol. Additionally, malicious parties identified during the protocol execution are added to the party's initial faulty list $\mathcal{F}_i$. The parties then return the updated faulty list $\mathcal{F}_i^\star$, with the notation $\mathcal{F}_i^\star$ used to distinguish it from the initial input faulty list $\mathcal{F}_i$.

Definition 3 (Correct or Detect Broadcast (*d-CoD*)). *Let Π be protocol executed by parties $P_1, ..., P_n$ where a designated sender P_s holds input $v \in \{0,1\}^*$. In addition, each party inputs a list of faulty parties $\mathcal{F}_i \subset \mathcal{P}$, and outputs a value $y_i \in \{0,1\}^*$, an updated faulty list $\mathcal{F}_i^\star \subset \mathcal{P}$, and a flag $det_i \in \{C, D\}$ upon terminating. Π achieves* Correct or Detect Broadcast (CoD)*, if the following properties hold whenever at most t parties are corrupted and for all honest parties P_i, $\mathcal{F}_i$ contains only corrupted parties.*

– *$\mathcal{F}$-soundness: If an honest party P_i outputs $\mathcal{F}_i^\star$, then $\mathcal{F}_i^\star$ consists only of corrupted parties. Furthermore, $\mathcal{F}_i \subseteq \mathcal{F}_i^\star$.*
– *Consistency: If $det_i = C$ for some honest party P_i, then every honest party P_j outputs $y_j = y_i$. In this case, we say that the protocol has* correctness.
– *Validity: If P_s is honest and is not included in $\mathcal{F}_j$ for every other honest party P_j and inputs v, then every honest party P_i outputs $(y_i = v, \mathcal{F}_i = \emptyset, det_i = C)$.*
– *d-Detection: If for some honest party P_i, $det_i = D$, then an additional d parties are added to the faulty lists of all honest parties; that is, $\left|\left(\bigcap_{P_j \in \mathcal{H}} \mathcal{F}_j^\star\right) \setminus \left(\bigcap_{P_j \in \mathcal{H}} \mathcal{F}_j\right)\right| \geq d$. In this case, we say that the protocol has* detection.
– *Termination: Every honest party terminates.*

Definition 4 ((0, 1)-Graded *d*-Detecting Agreement). *Let Π be a protocol executed by parties $P_1, ..., P_n$, where each party P_i inputs $v_i \in \{0,1\}$ and a list of faulty parties $\mathcal{F}_i \subset \mathcal{P}$ and outputs a value $y_i \in \{0,1\}$, a grade $g_i \in \{0,1\}$, and an updated faulty list $\mathcal{F}_i^\star \subset \mathcal{P}$ upon terminating. Π achieves* (0, 1)-Graded d-Detecting Agreement *if the following properties hold whenever at most t parties are corrupted and for all honest parties P_i, $\mathcal{F}_i$ contains only corrupted parties.*

– *Graded Validity: If all honest parties P_i have the same input value $v_i = \mathsf{v}$ then all honest parties output $y_i = \mathsf{v}$ and $g_i = 1$*
– *Graded Consistency: If two honest parties P_i and P_j output $g_i = g_j = 1$, respectively, then $y_i = y_j$*

- *d-Detection: If two honest parties P_i and P_j output $y_i = 1$ and $y_j = 0$, respectively, then an additional d parties are added to the faulty lists of all honest parties; that is, $\left|\left(\bigcap_{P_j \in \mathcal{H}} \mathcal{F}_j^\star\right) \setminus \left(\bigcap_{P_j \in \mathcal{H}} \mathcal{F}_j\right)\right| \geq d$.*
- *Soundness: If an honest party P_i outputs $\mathcal{F}_i^\star$, then $\mathcal{F}_i^\star$ consists only of corrupted parties. Furthermore, $\mathcal{F}_i \subseteq \mathcal{F}_i^\star$*
- *Termination: Every honest party terminates.*

Definition 5 ($(0,1,2)$-Graded d-Detecting Agreement). *Let Π be a protocol executed by parties $P_1, ..., P_n$, where each party P_i inputs $v_i \in \{0,1\}$ and a list of faulty parties $\mathcal{F}_i \subset \mathcal{P}$ and outputs a value $y_i \in \{0,1,\perp\}$, a grade $g_i \in \{0,1,2\}$, and an updated faulty list $\mathcal{F}_i^\star \subset \mathcal{P}$ upon terminating. A protocol Π achieves* $(0,1,2)$-Graded d-Detecting Agreement *if the following properties hold whenever at most t parties are corrupted and for all honest parties P_i, $\mathcal{F}_i$ contains only corrupted parties.*

- *Graded Validity: If all honest parties P_i have the same input value $v_i = \mathsf{v}$ then all honest parties output $y_i = \mathsf{v}$ and $g_i = 2$*
- *Graded Consistency: Let P_i and P_j denote honest parties that output y_i, g_i and y_j, g_j, respectively. Then (1) $|g_i - g_j| \leq 1$ and (2) $g_i, g_j \geq 1$ implies that $v_i = v_j$*
- *d-Detection: If any honest party P_i outputs $g_i < 2$, then an additional d parties are added to the faulty lists of all honest parties; that is, $\left|\left(\bigcap_{P_j \in \mathcal{H}} \mathcal{F}_j^\star\right) \setminus \left(\bigcap_{P_j \in \mathcal{H}} \mathcal{F}_j\right)\right| \geq d$.*
- *Soundness: If an honest party P_i outputs $\mathcal{F}_i^\star$, then $\mathcal{F}_i^\star$ consists only of corrupted parties. Furthermore, $\mathcal{F}_i \subseteq \mathcal{F}_i^\star$.*
- *Termination: Every honest party terminates.*

Definition 6 (Proof of Participation). *A* proof of participation PoP_i *for a party $P_i \in \mathcal{P}$ is a collection of $t+1$ signatures of the form $\langle P_i \rangle_{j_l}$ from distinct signers $P_{j_1}, \ldots, P_{j_{t+1}} \in \mathcal{P}$. We say that* PoP_i *is* valid *if for all $l \in [t+1]$, $\langle P_i \rangle_{j_l}$ is valid with respect to pk_{j_l}.*

Definition 7 (Signature Chain). *Let $m \in \{0,1\}^*$, let $k \in \mathbb{N}$, and let PoP_k be the proof of participation of party P_k as per Definition 6. We write $\langle m \rangle_\sigma$ to denote the nested messages and signatures $\langle \ldots \langle \langle m, \mathsf{PoP}_{j_1} \rangle_{j_1}, \mathsf{PoP}_{j_2} \rangle_{j_2}, \ldots \mathsf{PoP}_{j_k} \rangle_{j_k}$, where $j_1, \ldots j_k$ are distinct values in $[n]$, and refer to σ as a* signature chain of length k*. The expression $\langle m \rangle_\sigma$ is said to be* valid *if, for all k, the signature with respect to pk_{j_k} is valid and the proof of participation PoP_{jk} is valid.*

3 Deterministic Early-Stopping Byzantine Agreement

As previously discussed, both of our early-stopping protocols are built upon the $(0,1)$-Graded d-Detecting Byzantine Agreement protocol, which is itself derived from the Correct-or-Detect Broadcast protocol $\Pi_{\mathsf{d\text{-}CoD}}$ [11]. This protocol also utilizes the Proof of Participation protocol Π_{PoP} [11] as a subroutine. We adopt a bottom-up approach, initially introducing the aforementioned subroutines and subsequently demonstrating the construction of the $(0,1)$-Graded d-Detecting Agreement protocol and our early-stopping protocols.

3.1 Proof of Participation ($\varPi_{\mathsf{PoP}}$)

At a high level, the Proof of Participation protocol, $\varPi_{\mathsf{PoP}}$, allows each party to obtain a proof PoP of its honesty. A proof of participation, PoP, is considered *valid* if it consists of t+1 valid signatures from distinct parties $P_j \in \mathcal{P}$ in the form $\langle P_i \rangle_j$. To generate such a proof, each party P_i executes $\varPi_{\mathsf{PoP}}$ with the input $\mathcal{F}_i$, which represents its current view of faulty parties. In the first round of $\varPi_{\mathsf{PoP}}$, each party sends a message to all parties not in its faulty list $\mathcal{F}_i$, asserting their honesty. If a party P_j receives at least $t+1$ such messages, it uses them to construct its PoP proof.

Next, we define the two primary properties of $\varPi^k_{\mathsf{PoP}}$ in Lemma 1 and Lemma 2 (Fig. 1).

Protocol $\varPi_{\mathsf{PoP}}$

- **Input and Initialization:** Let $\mathcal{F}_i$ denote P_i's input. P_i sets $\mathsf{PoP}_i := \perp$
- **Round** 1**:**
  - For each party $P_j \notin \mathcal{F}_i$, party P_i sends $\langle P_j \rangle_i$ to party P_j
- **Output Determination:** If P_i receives valid signatures $\langle P_i \rangle_j$ from at least $t+1$ distinct parties, P_i collects these messages into PoP_i. P_i outputs PoP_i and terminates.

Fig. 1. Code of $\varPi_{\mathsf{PoP}}$ for party P_i.

Lemma 1. *Assume no honest party P_j is in the faulty list $\mathcal{F}_i$ of any other honest party P_i. Then, each honest party P_j outputs a valid* PoP_j.

Proof. There are at most $t < n/2$ malicious parties. Each honest party P_i sends $\langle P_j \rangle_i$ to every party $P_j \notin \mathcal{F}_i$. As per assumption, every honest party p_i will receive at least $t+1$ messages of $\langle p_i \rangle_j$. Consequently, every honest party sets its output PoP_i to the aggregation of those received messages. □

Lemma 2. *Assume there exists some party P_j such that $P_j \in \mathcal{F}_i$ for all honest parties $P_i \in \mathcal{P}$. Then, P_j does not output a valid* PoP_j.

Proof. There are at most $t < n/2$ malicious parties. No honest party will send $\langle p_j \rangle_i$ to $P_j \in \mathcal{F}_i$. Thus, P_j can collect at most $t < n/2$ such messages, which are not enough to form PoP_j. □

3.2 Correct or Detect Broadcast Protocols ($\varPi_{\mathsf{d\text{-}CoD}}$ and $\varPi_{\mathsf{d\text{-}MCoD}}$)

In essence, $\varPi_{\mathsf{d\text{-}CoD}}$ (Fig. 2) is a broadcast protocol that ensures either all parties agree on the sender's value, or all honest parties identify a common set of d corrupted parties. The protocol $\varPi_{\mathsf{d\text{-}CoD}}$ is 1-biased, meaning the designated sender P_s sends his value to parties only if it is $v_s = 1$; otherwise, he refrains from sending anything. Essentially, $\Pi_{\mathsf{d\text{-}CoD}}$ is a modified binary version of Dolev-Strong [9] that is 1- biased and forced to terminate in $d+5$ rounds. In the first round, all parties run protocol Π_{PoP} to obtain a valid PoP. Only parties with valid PoP are allowed to participate in the protocol. Every party tags along its PoP when sending a message and only accepts messages from parties if they tag along their valid PoP. In every round $r > 1$, if a party P_i receives a valid chain $\langle 1\rangle_\sigma$ with respect to Definition 7, including the sender's signature for the first time, it accepts the message, appends its own signature and PoP, and forwards it to all parties in the next round. Let r_i be the first round where party P_i receives such a message. P_i sets $y_i \in \{0,1\}$ and $det_i \in \{C, D\}$ based on the value of r_i. If $r_i \leq d+2$ or $d+5$, P_i outputs $det_i = C$; otherwise, it outputs $det_i = D$. If $r_i \leq d+3$, it outputs $y_i = 1$; otherwise, it outputs $y_i = 0$. For completeness, we show the $\Pi_{\mathsf{d\text{-}CoD}}$ protocol in Fig. 2 and state the correctness lemma (Lemma 3) for $\Pi_{\mathsf{d\text{-}CoD}}$. We provide the complete proof in the appendix.

Protocol $\Pi_{\mathsf{d\text{-}CoD}}$

- **Input and Initialization:** If $P_i = P_s$, let v_i and $\mathcal{F}_i$ denote $P_i's$ input. Otherwise, let $\mathcal{F}_i$ denote P_i's input.
  P_i sets $\mathcal{F}_i^\star := \emptyset$, $y_i := 0$, $det_i := C, r_i := d+5$.
- **Round 1:**
  - Party P_i runs Π_{PoP} on input $\mathcal{F}_i$. Let PoP_i denote the output.
- **Round 2 ($P_i = P_s$):** If the sender's initial value is $v_s = 1$, it sends $\langle 1, \mathsf{PoP}_s\rangle_s$ to all parties. (Otherwise, it does nothing.)
- **Rounds $r = 3$ to $d+5$ ($P_i \neq P_s$):**
  - If P_i received a valid signature chain $\langle 1\rangle_\sigma$ of length $r-1$ in the previous round and $r_i = r-2$, it appends to the chain its signature and PoP, i.e., it computes $m := \langle\langle 1\rangle_\sigma, \mathsf{PoP}_i\rangle_i$ and sends m to all parties.
  - If P_i receives a valid signature chain $\langle 1\rangle_\sigma$ of length r and $r_i = d+5$, it sets $r_i := r-1$. Furthermore, for $\langle 1\rangle_\sigma = \langle \ldots \langle\langle 1, \mathsf{PoP}_s\rangle_s, \ldots \mathsf{PoP}_k\rangle_k, \mathsf{PoP}_j\rangle_j$, P_i adds every party in $P_s, \ldots, P_k$ in the signature chain to $\mathcal{F}_i^\star$.
- **Output Determination:** If $P_i = P_s$, P_i sets $y_i := v_s, det_i := C$ and terminates. Else if $r_i \leq d+3$, party P_i sets $y_i := 1$. Else if $d+4 \leq r_i \leq d+5$, it sets $det_i := D$. P_i sets $\mathcal{F}_i^\star = \mathcal{F}_i^\star \cup \mathcal{F}_i$ Finally, P_i outputs $y_i, det_i, \mathcal{F}_i^\star$ and terminates.

Fig. 2. Code of $\Pi_{\mathsf{d\text{-}CoD}}$ for party P_i.

Lemma 3. *$\Pi_{\mathsf{d\text{-}CoD}}$ achieves d-CoD as per Definition 3 in $d+5$ rounds.*

Next, we construct a protocol, $\Pi_{\mathsf{d\text{-}MCoD}}$ (see Fig. 3), that extends the binary input range of $\Pi_{\mathsf{d\text{-}CoD}}$ to a multivalued range. To achieve this, multiple $\Pi_{\mathsf{d\text{-}CoD}}$ protocols can be executed concurrently, allowing the sender to send each bit of their message string. Due to the concurrent execution, the resultant protocol still runs in $d + 5$ rounds; however, the communication complexity increases proportionally with the input size.

For a party p_i to output $det_i = C$, all concurrently invoked $\Pi_{\mathsf{d\text{-}CoD}}$ instances must terminate with $det_i = C$. Otherwise, the party outputs $det_i = D$. The output value y_i is obtained by concatenating all output bits from each $\Pi_{\mathsf{d\text{-}CoD}}$ instance. The output faulty list $\mathcal{F}_i^\star$ is the union of all faulty lists produced by each invoked instance of $\Pi_{\mathsf{d\text{-}CoD}}$. □

Protocol $\Pi_{\mathsf{d\text{-}MCoD}}$

- **Input and Initialization:** If $P_i = P_s$, let v_s and $\mathcal{F}_i$ denote P_s's input and P_i sets $l := |v_s|$. Otherwise, let $\mathcal{F}_i$ denote P_i's input. P_i sets $\mathcal{F}_i^\star := \emptyset, y_i := \perp, det_i := D, r_i := d + 5$
- **Rounds** $r = 1$ **to** $d + 5$**:**
  - Party P_s invokes in parallel l instances of $\Pi_{\mathsf{d\text{-}CoD}}$, where the input for the jth instance is bit $v_i[j], j \in [l]$. Let $(y^j_{CoD}, det^j_i, \mathcal{F}^j_i)$ denote the output of the jth instance for party P_i
- **Output Determination:** If $det^j_i = C$ for all $j \in [l]$, P_i sets $det_i := C$. It sets $y_i := y^1_{CoD}, \ldots, y^l_{CoD}$ and $\mathcal{F}_i^\star = \bigcup_{j=1}^{l} \mathcal{F}^j_i$. P_i outputs $(y_i, det_i, \mathcal{F}_i)$ and terminates.

Fig. 3. Code of $\Pi_{\mathsf{d\text{-}MCoD}}$ for party P_i.

In the following lemma, we prove the correctness of $\Pi_{\mathsf{d\text{-}MCoD}}$ per Definition 3

Lemma 4. *$\Pi_{\mathsf{d\text{-}MCoD}}$ achieves d-CoD as per Definition 3 and terminates in $d+5$ rounds.*

Proof. Assume that for each honest party P_i, $P_i \notin \mathcal{F}_j$ for any honest party P_j.

$\mathcal{F}$-soundness: The output faulty list $\mathcal{F}_i^\star$ is the union of all faulty lists $\mathcal{F}^j_i$ produced by the l parallel invocations of $\Pi_{\mathsf{d\text{-}CoD}}$. Based on the $\mathcal{F}$-soundness of the $\Pi_{\mathsf{d\text{-}CoD}}$ protocol, the resulting faulty list $\mathcal{F}_i^\star$ contains only malicious parties.

Consistency: If an honest party P_i outputs $det_i = C$, then for each $j \in [l]$, $det^j_i = C$. Thus, by consistency of $\Pi_{\mathsf{d\text{-}CoD}}$, each party P_j outputs the same bits in each of the l parallel instances of $\Pi_{\mathsf{d\text{-}CoD}}$ as party P_i. Since the output y_i is the concatenation of all output bits of the l instances of $\Pi_{\mathsf{d\text{-}CoD}}$, party P_j will output $y_j = y_i$.

Validity: If P_s is honest, it follows the same logic as discussed earlier since the output value y_i is simply the concatenation of the output values of all invoked $\Pi_{\mathsf{d\text{-}CoD}}$ instances and $det_i = C$ holds if for each instance j among those instances, $det^j_i = C$. Thus, validity follows directly from validity of $\Pi_{\mathsf{d\text{-}CoD}}$.

d-Detection: For a party to output $det_i = D$, at least one instance $j \in [l]$ among the l parallel instances of $\Pi_{\mathsf{d\text{-}CoD}}$ output $det^j_i = D$. Thus, the d-Detection property of $\Pi_{\mathsf{d\text{-}CoD}}$ implies that at least d malicious parties are added to every honest party P_i's faulty list $\mathcal{F}_i^\star$ via $\mathcal{F}_i^j$.

Termination: $\Pi_{\mathsf{d\text{-}MCoD}}$ consists of concurrent instances of $\Pi_{\mathsf{d\text{-}CoD}}$. Based on the assumption that $\Pi_{\mathsf{d\text{-}CoD}}$ terminates, $\Pi_{\mathsf{d\text{-}MCoD}}$ will also terminate.

Round Complexity. $\Pi_{\mathsf{d\text{-}MCoD}}$ consists of concurrent execution of $\Pi_{\mathsf{d\text{-}CoD}}$, which runs in $d+5$ rounds. □

3.3 (0, 1)-Graded d-Detecting Agreement Construction ($\Pi_{\mathsf{1\text{-}GDA}}$)

In summary, $\Pi_{\mathsf{1\text{-}GDA}}$ (see Fig. 4) is a variant of Graded Consensus protocols [10]. However, in $\Pi_{\mathsf{1\text{-}GDA}}$, honest parties also output a list of detected malicious parties. $\Pi_{\mathsf{1\text{-}GDA}}$ ensures that either every honest party outputs the same value y_i, or every honest party identifies at least d malicious parties (achieving d-detection).

In $\Pi_{\mathsf{1\text{-}GDA}}$, each party has input $v_i \in \{0,1\}$ and faulty list $\mathcal{F}_i$. Each party outputs a value $y_i \in \{0,1\}$, a grade $g_i \in \{0,1\}$, and an updated list of identified malicious parties $\mathcal{F}_i^\star \subset \mathcal{P}$. In the first round, each party P_i invokes $\Pi_{\mathsf{d\text{-}MCoD}}$ with input (i, v_i). The reason for sending i along with the initial variable is that the initial variable could be 0. Due to $\Pi_{\mathsf{d\text{-}MCoD}}$'s construction, the designated sender will not send anything if the initial variable is 0, meaning parties will not receive anything from the sender to determine if the party was honest or not. Therefore, a unified message is sent so that if a party sent $i \neq \perp$, all honest parties will consider it honest and take its value into consideration when calculating the final output. It is considered honest due to only parties with valid PoP can send messages according to the construction of $\Pi_{\mathsf{d\text{-}MCoD}}$. For simplicity, we denote $\Pi^j_{\mathsf{d\text{-}MCoD}}$ as the protocol instance where P_j is the sender. Each party stores the output $((i_{i,j}, y_{i,j}), det_i^j, \mathcal{F}_i^j)$ from all terminated instances of $\Pi^j_{\mathsf{d\text{-}MCoD}}$ for each $P_j \in \mathcal{P}$. Consequently, party P_i maintains a list H_i of all parties P_j that sent a valid $i \neq \perp$ via $\Pi^j_{\mathsf{d\text{-}MCoD}}$. Each party P_i takes the union of all the faulty lists output by all $\Pi_{\mathsf{d\text{-}MCoD}}$ instances to form $\mathcal{F}_i^\star$, in addition to the parties in its initial faulty list $\mathcal{F}_i$.

To determine the output value y_i and grade g_i, a party P_i only considers the output of $\Pi^j_{\mathsf{d\text{-}MCoD}}$ from parties P_j in H_i. If there is a bit $\mathsf{v} \in \{0,1\}$ such that for at least $t+1$ of the parties $P_j \in H_i$, $y^j_{mCoD} = \mathsf{v}$ and $det_i^j = C$, party P_i sets its output to $y_i = \mathsf{v}$ and $g_i = 1$. Otherwise, if no such $t+1$ parties exist, P_i outputs the majority value over values $y_{i,j}$ among parties P_j in H_i. The protocol runs for $d+5$ rounds due to $\Pi_{\mathsf{d\text{-}MCoD}}$. We proceed to prove the correctness of $\Pi_{\mathsf{1\text{-}GDA}}$.

Lemma 5. $\Pi_{\mathsf{1\text{-}GDA}}$ *achieves graded validity as per Definition 4.*

Proof. Assume that for all honest parties P_i, $v_i = \mathsf{v}$. Further, assume that for each honest party P_i, $P_i \notin \mathcal{F}_j$ for any honest party P_j. In the first round, each honest party P_i invokes as the sender, $\Pi_{\mathsf{d\text{-}MCoD}}$ on input $((i, \mathsf{v}), \mathcal{F}_i)$. According to the validity of $\Pi_{\mathsf{d\text{-}MCoD}}$ (Definition 3), if P_i is honest, each honest party P_j outputs $det_j^i = C$, $i_{j,i} = i$ and $y_{j,i} = v$. Thus, every honest P_i will add P_j to the

Protocol $\Pi_{\text{1-GDA}}$

- **Input and Initialization:** Let v_i and $\mathcal{F}_i$ denote P_i's input. P_i sets $y_i := v_i, g_i := 0, H_i, \mathcal{F}_i^\star := \emptyset$, and $y_{i,j}, i_{i,j} := \perp$ for $j \in [n]$
- **Rounds** $r = 1$ **to** $d + 5$**:**
  - Party P_i invokes $\Pi_{\text{d-MCoD}}$ on input $((i, v_i), \mathcal{F}_i)$. Denote the instance of $\Pi_{\text{d-MCoD}}$ in which P_j is the sender as $\Pi^j_{\text{d-MCoD}}$.
  - Party P_i stores the output of $\Pi^j_{\text{d-MCoD}}$ for $P_j \in \mathcal{P}$; $((i_{i,j}, y_{i,j}), det_i^j, \mathcal{F}_i^j) := \Pi^j_{\text{d-MCoD}}$
  - For each party P_j, if $i_{i,j}$ is j, P_i adds P_j to H_i.
- **Output Determination:**
  - Party P_i accumulates the faulty lists of all instances of $\Pi^j_{\text{d-MCoD}}$ along with the input $\mathcal{F}_i$ as $\mathcal{F}_i^\star = \bigcup_{j\in[n]} \mathcal{F}_i^j \cup \mathcal{F}_i$.
  - If there exists $\mathsf{v} \in \{0, 1\}$ and at least $t + 1$ instances of $\Pi^j_{\text{d-MCoD}}$ for $P_j \in H_i$ that terminate with $det_i^j = C$ and output $y_{i,j} = \mathsf{v}$, then P_i sets the output value $y_i := \mathsf{v}$ and the grade $g_i := 1$.
  - Otherwise, P_i sets $y_i := \mathsf{v}$, where v is the the majority bit among values $y_{i,j}$ where $P_j \in H_i$.
  - Party P_i outputs $y_i, g_i, \mathcal{F}_i^\star$ and terminates.

Fig. 4. Code of $\Pi_{\text{1-GDA}}$ for party P_i.

list H_i. Thus, since there are at most $t < n/2$ malicious parties, every honest party P_i will output $(\mathsf{v}, C, \mathcal{F}_i^j)$ from at least $t+1$ instances of $\Pi_{\text{d-MCoD}}$ for parties $P_j \in H_i$. Consequently, each honest party sets $y_i = \mathsf{v}$ and $g_i = 1$. □

Lemma 6. $\Pi_{\text{1-GDA}}$ *achieves graded consistency as per Definition 4.*

Proof. Assume that for each honest party P_i, $P_i \notin \mathcal{F}_j$ for any honest party P_j. A party P_i outputs $y_i = \mathsf{v}$ and $g_i = 1$ if at least $t + 1$ instances $\Pi^j_{\text{d-MCoD}}$ corresponding to parties $P_j \in H_i$ terminate with $det_i^j = C$, and have the same output value $y_{i,j} = \mathsf{v}$. From consistency of $\Pi_{\text{d-MCoD}}$, every other honest party P_j outputs $i_{j,k} = i$ and $y_{j,k} = \mathsf{v}$ for the same instances and adds the corresponding parties to those instances to H_j. Since $t < n/2$, the majority bit over all values $y_{j,k}, k \in H_j$ is also equal to v for every honest party P_j. Consequently, each honest party sets $y_i = \mathsf{v}$.

We proceed to prove the d-detection property.

Lemma 7. $\Pi_{\text{1-GDA}}$ *achieves d-detection as per Definition 4.*

Proof. Assume that for each honest party P_i, $P_i \notin \mathcal{F}_j$ for any other honest party P_j. Suppose two honest parties, P_i and P_j, output different values $y_i \neq y_j$ along with respective grades $g_i = g_j = 0$ and faulty lists $\mathcal{F}_i^\star$ and $\mathcal{F}_j^\star$. P_i determines y_i as the majority bit over values $y_{i,j}$ output from $\Pi^j_{\text{d-MCoD}}$ where $P_j \in H_i$. The majority of these values can only differ if an instance of $\Pi^k_{\text{d-MCoD}}$ outputs different values $y_{i,k} \neq y_{j,k}$ for P_i and P_j. The d-detection property of $\Pi_{\text{d-MCoD}}$

ensures that at least d malicious parties are added to the faulty list of every honest party when they take the union of the faulty lists output in all instances of $\Pi^{k}_{\mathsf{d\text{-}MCoD}}$.

Finally, we prove soundness and termination.

Lemma 8. *$\Pi_{\mathsf{1\text{-}GDA}}$ achieves soundness, and termination as per Definition 4.*

Proof. Assume that for each honest party P_i, $P_i \notin \mathcal{F}_j$ for any honest party P_j.

Soundness. An honest party P_i adds additional parties to its initial faulty list $\mathcal{F}_i$ by including the parties from the union of all the faulty lists generated by the $\Pi^{j}_{\mathsf{d\text{-}MCoD}}$ instances for each $P_j \in \mathcal{P}$. According to the $\mathcal{F}$-soundness of $\Pi_{\mathsf{d\text{-}MCoD}}$, the resulting $\mathcal{F}^{\star}_i$ will only include malicious parties.

Termination. $\Pi_{\mathsf{1\text{-}GDA}}$ is constructed from concurrent instances $\Pi_{\mathsf{d\text{-}MCoD}}$. Based on the assumption that $\Pi_{\mathsf{d\text{-}MCoD}}$ terminates, $\Pi_{\mathsf{1\text{-}GDA}}$ will also terminate.

Round Complexity. $\Pi_{\mathsf{1\text{-}GDA}}$ protocol runs for $d+5$ rounds as $\Pi_{\mathsf{d\text{-}MCoD}}$ runs for $d+5$ rounds. □

We summarize the previous lemmata into the main following Lemma of this section:

Lemma 9. *$\Pi_{\mathsf{1\text{-}GDA}}$, (Fig. 4) achieves $(0,1)$-Graded d-Detecting Agreement as per Definition 5. Furthermore, $\Pi_{\mathsf{1\text{-}GDA}}$ terminates in $d+5$ rounds.*

3.4 Deterministic Early-Stopping Byzantine Agreement Protocol (Π_{BA^d})

In this subsection, we demonstrate how to construct the deterministic early-stopping Byzantine agreement protocol, Π_{BA^d}, using $\Pi_{\mathsf{1\text{-}GDA}}$. In Π_{BA^d}, each party starts with an input value $v_i \in \{0,1\}$ and outputs an output value $y_i \in \{0,1\}$. Π_{BA^d} runs in iterations. In each iteration k, parties run $\Pi_{\mathsf{1\text{-}GDA}}$ with input $(v_i, \mathcal{F}_i)$. Consequently, each party P_i stores the output $(y_{\mathsf{GDA_i}}, g_i, \mathcal{F}_i)$ of $\Pi_{\mathsf{1\text{-}GDA}}$.

Based on the grade g_i obtained from $\Pi_{\mathsf{1\text{-}GDA}}$, each party P_i determines whether it is safe to terminate. If P_i outputs $g_i = 0$, it indicates that it is not safe to terminate, and more iterations are required. P_i updates its input value for the next iteration based on the output value $y_{\mathsf{GDA_i}} \in \{0,1\}$ of $\Pi_{\mathsf{1\text{-}GDA}}$, setting $v_i = y_{\mathsf{GDA_i}}$. Conversely, if P_i outputs $g_i = 1$, it is confident that all other honest parties P_j output the same value $y_{\mathsf{GDA_i}} = y_{\mathsf{GDA_j}}$ due to the graded consistency of $\Pi_{\mathsf{1\text{-}GDA}}$. In this case, P_i runs for one more iteration to ensure that other honest parties can also safely terminate on the same value, as proven in Lemma 13. Note, a party can set its output value y_i in iteration k, but terminates a few iterations later. A party only terminates when $\mathsf{halt}_i = \mathit{true}$.

Each iteration consists of $d+5$ rounds: $d+5$ rounds for $\Pi_{\mathsf{1\text{-}GDA}}$. Therefore, the overall round complexity of Π_{BA^d} depends on the number of iterations it runs. We demonstrate in Lemma 14 that the number of iterations is a function of f.

```
Protocol $\Pi_{\mathsf{BA}^d}$

– Input and Initialization: Let $v_i$ denote $P_i$'s input. $P_i$ sets $\mathsf{halt}_i :=$
  $\mathsf{false}, y_i, y_{\mathsf{GDA_i}} := \perp, \mathsf{wait}_i := \infty$ $\mathcal{F}_i := \emptyset$
– While $\mathsf{halt}_i = \mathsf{false}$ do
    • Rounds $1$ to $d+5$:
        ∗ $P_i$ runs protocol $\Pi_{\mathsf{1\text{-}GDA}}$ with input $(v_i, \mathcal{F}_i)$ and stores output
          $(y_{\mathsf{GDA_i}}, g_i, \mathcal{F}_i) := \Pi_{\mathsf{1\text{-}GDA}}$
        ∗ If $\mathsf{wait}_i = 1$, $P_i$ sets $\mathsf{halt}_i := true$. Otherwise, if $g_i = 1$ and
          $\mathsf{wait}_i > 1$, $P_i$ sets $\mathsf{wait}_i := 1$ and $y_i := y_{\mathsf{GDA_i}}$
        ∗ Each party $P_i$ updates the input of next iteration by setting
          $v_i := y_{\mathsf{GDA_i}}$
– Output Determination: If $\mathsf{halt}_i = \mathsf{true}$, $P_i$ outputs $y_i$ and terminates.
```

Fig. 5. Code of Π_{BA^d} for party P_i.

First, we establish that honest parties are never included in the faulty lists of other honest parties in any iteration. From this point forward, we assume this lemma holds indefinitely. Consequently, the assumption of $\Pi_{\mathsf{1\text{-}GDA}}$ as stated in Definition 4 is always valid, and we may omit it from proofs for simplicity.

Lemma 10. *At the start of each iteration k of Π_{BA^d}, the faulty list $\mathcal{F}_i$ of every honest party P_i contains only corrupted parties.*

Proof. In the first iteration $k = 1$, the faulty lists of all honest parties are empty, so the lemma holds trivially. For subsequent iterations $k > 1$, each party updates its $\mathcal{F}_i$ based on the output of $\Pi_{\mathsf{1\text{-}GDA}}$. According to the soundness property of $\Pi_{\mathsf{1\text{-}GDA}}$, no honest party P_i is included in the $\mathcal{F}_j$ of any other honest party P_j in any of these iterations. Thus the claim follows by a simple induction. □

Next, we prove that if all honest parties set $y_{\mathsf{GDA_i}}$ to the same value in iteration k, all honest parties terminate by at most iteration $k + 2$.

Lemma 11. *If all honest parties P_i set $y_{\mathsf{GDA_i}}$ to the same value in iteration k, then all honest parties terminate by at most iteration $k + 2$.*

Proof. Let all honest parties P_i set $y_{\mathsf{GDA_i}}$ to the same value v in iteration k, i.e., $y_{\mathsf{GDA_i}} := \mathsf{v}$. If a party has grade $g_i = 1$, it sets $\mathsf{wait}_i := 1$ and $y_i := \mathsf{v}$. Otherwise, it does nothing. Consequently, each party updates its input value v_i for the subsequent iteration based on this output value of $\Pi_{\mathsf{1\text{-}GDA}}$, such that $v_i := \mathsf{v}$. In the next iteration $(k + 1)$, all honest parties invoke $\Pi_{\mathsf{1\text{-}GDA}}$ with the same input value v. According to the validity of $\Pi_{\mathsf{1\text{-}GDA}}$, all honest parties set $g_i := 1$. If $\mathsf{wait}_i = 1$, P_i sets $\mathsf{halt}_i = \mathsf{true}$, outputs $y_i = \mathsf{v}$ and terminates in iteration $k + 1$. Otherwise, each other honest party sets $\mathsf{wait}_i = 1$ and $y_i := \mathsf{v}$. In iteration $k + 2$, as $\mathsf{wait}_i = 1$, each honest party sets $\mathsf{halt}_i := \mathsf{true}$, outputs $y_i = \mathsf{v}$ and terminates. □

Next, we proceed with proving validity and consistency for Π_{BA^d}.

Lemma 12. *Π_{BA^d} achieves validity per Definition 1*

Proof. Assume all honest parties have the same initial value ($v_i = \mathsf{v}$). Every party invokes $\Pi_{\mathsf{1\text{-}GDA}}$ with input $(\mathsf{v}, \mathcal{F}_i)$. From graded validity of $\Pi_{\mathsf{1\text{-}GDA}}$, every honest party outputs $y_{\mathsf{GDA_i}} = \mathsf{v}$ and $g_i = 1$. Consequently, as $g_i = 1$, every honest party sets $\mathsf{wait}_i := 1$ and $y_i = \mathsf{v}$. In the next iteration, each honest party outputs $y_i = \mathsf{v}$ and terminates. □

Lemma 13. *Π_{BA^d} achieves consistency per Definition 1*

Proof. Let P_i denote the first honest party to set $\mathsf{wait}_i := 1$ and $y_i := \mathsf{v}$ in the earliest iteration, say $k > 0$, indicating it will *wait* for one more iteration before terminating. This occurs when p_i sets its grade g_i to 1, determined by the output of $\Pi_{\mathsf{1\text{-}GDA}}$ in iteration k. According to the graded consistency of $\Pi_{\mathsf{1\text{-}GDA}}$, every other honest party P_j outputs $y_{\mathsf{GDA_j}} = \mathsf{v}$ in iteration k. According to Lemma 11, every honest party terminate with output $y_i = \mathsf{v}$ by the latest in iteration $k+2$ □

Finally, we prove the round complexity of Π_{BA^d}.

Lemma 14. *Π_{BA^d} terminates in $(d+5)\cdot(\lfloor f/d\rfloor + 3)$ rounds.*

Proof. In any iteration k, if honest parties P_i and P_j have the same output value $y_{\mathsf{GDA_i}} = y_{\mathsf{GDA_j}}$ based on the output of $\Pi_{\mathsf{1\text{-}GDA}}$, then all honest parties will terminate by iteration $k+2$, as proven in Lemma 11. If in some iteration, P_i and P_j have different output values, i.e., $y_{\mathsf{GDA_i}} \neq y_{\mathsf{GDA_j}}$ from $\Pi_{\mathsf{1\text{-}GDA}}$, then according to the d-detection property of $\Pi_{\mathsf{1\text{-}GDA}}$, at least d malicious parties are added to the faulty list $\mathcal{F}_i$ of all honest parties $P_i \in \mathcal{P}$. Thus, since there are f faulty parties, there can be at most $\lfloor f/d\rfloor$ many iterations where there are distinct honest parties P_i and P_j that output different values $y_{\mathsf{GDA_i}} \neq y_{\mathsf{GDA_j}}$ from $\Pi_{\mathsf{1\text{-}GDA}}$. Thus, after at most $\lfloor f/d\rfloor + 1$ many iterations, all honest parties output the same value $y_{\mathsf{GDA_i}}$. Hence, they all terminate by iteration $\lfloor f/d\rfloor + 3$ by Lemma 11. Since each iteration takes $d+5$ rounds, the overall complexity comes out to $(d+5)\cdot(\lfloor f/d\rfloor + 3))$. □

We sum up Lemmata 12, 13, and 14 into Theorem 1 as follows:

Theorem 1. *Assume a PKI setup and $t < n/2$. Π_{BA^d} (Fig. 5) achieves Byzantine Agreement per Definition 1. Furthermore, Π_{BA^d} terminates in $(d+5)\cdot(\lfloor f/d\rfloor + 3)$ rounds, for any execution with $f \leq t$ corrupted parties and runs in communication complexity $O(f\cdot n^4)$.*

Proof. Byzantine Agreement follows from the preceding lemmata. For the communication complexity, we note that the complexity of an instance of $\Pi_{\mathsf{d\text{-}CoD}}$ is $O(n^2\cdot d)$ and during each iteration of Π_{BA^d}, $O(n^2)$ such instances are called to broadcast the PoPs of length $O(n)$ bit by bit for $O(n)$ senders. Since the protocol has $O(f/d)$ iterations, the overall complexity is $O(n^2\cdot n^2\cdot d\cdot f/d) = O(n^4\cdot f)$.

4 Byzantine Agreement with Expected Constant and Worst-Case Early-Stopping Round Complexity

In this section, we introduce our randomized Byzantine Agreement protocol, Π_{BA^r}, which achieves both expected constant time and worst-case early-stopping round complexity. Similar to our deterministic protocol, Π_{BA^r} is built using the $(0,1,2)$-Graded d-Detecting Agreement protocol, $\Pi_{\mathsf{2\text{-}GDA}}$. Therefore, we begin by introducing $\Pi_{\mathsf{2\text{-}GDA}}$ and then present the complete construction of Π_{BA^r}.

4.1 $(0,1,2)$-Graded d-Detecting Agreement ($\Pi_{\mathsf{2\text{-}GDA}}$)

Similar to $\Pi_{\mathsf{1\text{-}GDA}}$ protocol, $\Pi_{\mathsf{2\text{-}GDA}}$ is a variant of Graded Consensus protocols [10], which allows honest parties to also output a list of detected malicious parties. In $\Pi_{\mathsf{2\text{-}GDA}}$, each party starts with $v_i \in \{0,1\}$ and faulty list $\mathcal{F}_i$. Each party outputs a value $y_i \in \{0,1,\perp\}$, a grade $g_i \in \{0,1\}$, and an updated list of identified malicious parties $\mathcal{F}_i^\star \subset \mathcal{P}$. $\Pi_{\mathsf{2\text{-}GDA}}$ is constructed from $\Pi_{\mathsf{1\text{-}GDA}}$ and the black-box $(0,1,2)$-Graded Agreement protocol from [12], $\Pi_{\mathsf{2\text{-}GA}}$, which we include in the Appendix. In the first round, each party P_i invokes $\Pi_{\mathsf{1\text{-}GDA}}$ with input $(v_i,\mathcal{F}_i)$, storing the resulting output $(y_i^\star, g_i^\star, \mathcal{F}_i^\star)$. To enhance the confidence on its output value, the parties run $\Pi_{\mathsf{2\text{-}GA}}$ with $y_i^\star$ as its input. Finally, party P_i terminates and outputs $(y_i, g_i, \mathcal{F}_i^\star)$, where they are the output of $\Pi_{\mathsf{2\text{-}GA}}$. Note that the output $\mathcal{F}_i^\star$ is the faulty list output from $\Pi_{\mathsf{1\text{-}GDA}}$ and does not get updated further. The protocol runs for $d+9$ rounds: $d+5$ for $\Pi_{\mathsf{1\text{-}GDA}}$ and 4 additional rounds for $\Pi_{\mathsf{2\text{-}GA}}$ (Fig. 6.

Protocol $\Pi_{\mathsf{2\text{-}GDA}}$

- **Input and Initialization:** Let v_i and $\mathcal{F}_i$ denote P_i's input. P_i sets $y_i, y_i^\star := \perp, g_i, g_i^\star := 0, \mathcal{F}_i^\star := \emptyset$
- **Rounds $r = 1$ to $d+5$:**
  - P_i invokes $\Pi_{\mathsf{1\text{-}GDA}}$ with input $(v_i,\mathcal{F}_i)$. Let $(y_i^\star, g_i^\star, \mathcal{F}_i^\star)$ denote the output.
- **Rounds $r = d+6$ to $r = d+9$:**
  - P_i invokes $\Pi_{\mathsf{2\text{-}GA}}$ with input $y_i^\star$ and let (y_i, g_i) denote the output.
- **Output Determination:** P_i outputs $(y_i, g_i, \mathcal{F}_i^\star)$ and terminates

Fig. 6. Code of $\Pi_{\mathsf{2\text{-}GDA}}$ for party P_i.

Lemma 15. *Assume $\Pi_{\mathsf{2\text{-}GA}}$ achieves* $(0,1,2)$*-Graded Agreement per Definition 2. $\Pi_{\mathsf{2\text{-}GDA}}$ achieves* $(0,1,2)$*-Graded Faulty-Detecting Byzantine Agreement per Definition 5.*

Proof. Assume that for each honest party P_i, $P_i \notin \mathcal{F}_j$ for any honest party P_j. Suppose that every honest party P_i inputs $(v_i,\mathcal{F}_i)$ to $\Pi_{\mathsf{2\text{-}GDA}}$, where $v_i \in \{0,1\}$ and $\mathcal{F}_i \subset \mathcal{P}$.

Graded Validity. By assumption, every honest party starts with $v_i = \mathsf{v}$, and invokes $\Pi_{\text{1-GDA}}$ with input $(\mathsf{v}, \mathcal{F}_i)$. According to the graded validity of $\Pi_{\text{1-GDA}}$ (Definition 4), all honest parties outputs $(\mathsf{v}, 1, \mathcal{F}_i^\star)$. Thus in round $d + 6$, every honest party invokes $\Pi_{\text{2-GA}}$ with input v. From graded validity of $\Pi_{\text{2-GA}}$, P_i outputs $y_i = \mathsf{v}$ and $g_i = 2$.

Graded Consistency. A party p_i sets its g_i and y_i based on the output of $\Pi_{\text{2-GA}}$. From graded consistency of $\Pi_{\text{2-GA}}$, this holds.

***d*-Detection.** Assume an honest party p_i outputs a $g_i < 2$. If an honest party p_i outputs a $g_i < 2$, it follows from graded validity of $\Pi_{\text{2-GA}}$ that not all parties input the same value to $\Pi_{\text{2-GA}}$. Parties invoke $\Pi_{\text{2-GA}}$ with the output value they obtained from $\Pi_{\text{1-GDA}}$, so there must be two honest parties P_i and P_j that output distinct values y_i^* and y_j^* from $\Pi_{\text{1-GDA}}$. Thus, d-detection of $\Pi_{\text{2-GDA}}$ is directly implied by d-detection of $\Pi_{\text{1-GDA}}$.

Soundness. The output faulty list, denoted as $\mathcal{F}_i^\star$, is based on the output faulty list from $\Pi_{\text{1-GDA}}$. Due to the soundness property of $\Pi_{\text{1-GDA}}$, $\mathcal{F}_i^\star$ contains only malicious parties.

Termination: The protocol invokes $\Pi_{\text{1-GDA}}$ and $\Pi_{\text{2-GA}}$, which terminates as per Definitions 4 and 2 respectively.

4.2 Byzantine Agreement with Expected Constant and Worst-Case Early-Stopping Round Complexity

In this subsection, we present our randomized Byzantine agreement protocol which has expected constant time and worst case early-stopping round complexity. We demonstrate how to construct the randomized early-stopping Byzantine agreement protocol, Π_{BA^r}, using $\Pi_{\text{2-GDA}}$. In Π_{BA^r}, each party starts with an input value $v_i \in \{0, 1\}$ and outputs an output value $y_i \in \{0, 1\}$. Π_{BA^r} runs in iterations. In each iteration k, parties run $\Pi_{\text{2-GDA}}$ with input $(v_i, \mathcal{F}_i)$. Consequently, each party P_i stores the output $(y_{\mathsf{GDA_i}}, g_i, \mathcal{F}_i)$ of $\Pi_{\text{2-GDA}}$. Based on the grade g_i obtained from $\Pi_{\text{2-GDA}}$, each party P_i determines whether it is safe to terminate. If P_i outputs $g_i < 2$, it indicates that it is not safe to terminate, and more iterations are run to reach agreement.

Conversely, if P_i outputs $g_i = 2$, it is confident that all other honest parties P_j output the same value $y_{\mathsf{GDA_i}} = y_{\mathsf{GDA_j}}$ due to the graded consistency of $\Pi_{\text{2-GDA}}$. Party P_i then updates its input value for the next iteration based on the output grade $g_i \in \{0, 1, 2\}$ of $\Pi_{\text{2-GDA}}$. If $g_i > 0$, it updates its input value to the next iteration based on the output value $y_{\mathsf{GDA_i}} \in \{0, 1\}$ of $\Pi_{\text{2-GDA}}$, setting $v_i = y_{\mathsf{GDA_i}}$. Otherwise, if $g_i = 0$, it sets its input value to the next iteration based on the random coin it receives from the CoinFlip protocol. We show in Lemma 20 that Π_{BA^r} has expected constant time.

Each iteration consists of $d + 9$ rounds due to the $\Pi_{\text{2-GDA}}$ protocol. Therefore, the overall round complexity of Π_{BA^r} depends on the number of iterations it runs in the worst case. We demonstrate in Lemma 14 that the number of iterations in the worst case is a function of f.

Protocol Π_{BA^r}

- **Input and Initialization:** Let v_i denote P_i's input. P_i sets $k := 0$ $\mathsf{halt}_i := \mathsf{false}, y_i, y_{\mathsf{GDA_i}} := \perp, \mathsf{wait}_i := \infty$ $\mathcal{F}_i := \emptyset$
- **While** $\mathsf{halt}_i = \mathsf{false}$ **do**
  - $k := k + 1$
  - **Rounds** 1 **to** $d + 9$**:**
    - $*$ P_i invokes protocol $\Pi_{\mathsf{2\text{-}GDA}}$ with input $(v_i, \mathcal{F}_i)$. Let $(y_{\mathsf{GDA_i}}, g_i, \mathcal{F}_i)$ denote the output.
    - $*$ P_i updates the input for next iteration $v_i := y_{\mathsf{GDA_i}}$
    - $*$ If $g_i = 2$ and $\mathsf{wait}_i = 1$, P_i sets $\mathsf{halt}_i := \mathsf{true}$. Otherwise, if $g_i = 2$ and $\mathsf{wait}_i > 1$, P_i sets $\mathsf{wait}_i = 1$ and $y_i = y_{\mathsf{GDA_i}}$
    - $*$ If $g_i = 0$, party P_i updates the next iteration's input using the common coin, $c_i^{(k)} \leftarrow \mathsf{CoinFlip}(k)$. It sets $v_i := c_i^{(k)}$.
- **Output Determination:** If $\mathsf{halt}_i = \mathsf{true}$, P_i outputs y_i and terminates.

Fig. 7. Code of Π_{BA^r} for party P_i

Similar to $\Pi_{\mathsf{1\text{-}GDA}}$, we also establish that honest parties are never included in the faulty lists of other honest parties in any iteration, which is a needed assumption for $\Pi_{\mathsf{2\text{-}GDA}}$

Lemma 16. *At the start of each iteration, the faulty list $\mathcal{F}_i$ of every honest party P_i contains only corrupted parties.*

Proof. The proof follows from Lemma 10 and from the fact that $\mathcal{F}_i$ in Π_{BA^r} is based on the faulty list produced by $\Pi_{\mathsf{2\text{-}GDA}}$. □

We proceed to prove both validity of Π_{BA^r}.

Lemma 17. *Π_{BA^r} achieves validity per Definition 1*

Proof. Assume all honest parties have the same initial value ($v_i = \mathsf{v}$). Every party invokes $\Pi_{\mathsf{2\text{-}GDA}}$ in the second round with input $(v_i, \mathcal{F}_i)$. From graded validity of $\Pi_{\mathsf{2\text{-}GDA}}$, every honest party outputs $(y_{\mathsf{GDA_i}} = \mathsf{v}, g_i = 2, \mathcal{F}_i)$. Consequently, as $g_i = 2$, every honest party sets $\mathsf{wait}_i := 1$ and $y_i = \mathsf{v}$. In the next iteration, each honest party outputs $y_i = \mathsf{v}$ and terminates. □

We state the following lemma that will help us in proving consistency.

Lemma 18. *If all honest parties set $y_{\mathsf{GDA_i}}$ to the same value in iteration k, then all honest parties will terminate by at most iteration $k + 2$.*

Proof. The proof follows similar logic to Lemma 11. □

Next, we prove consistency.

Lemma 19. *Π_{BA^d} achieves consistency per Definition 1*

Proof. Let P_i be the first honest party to set $\mathsf{wait}_i = 1$ and $y_i := \mathsf{v}$ in the earliest iteration, say $k > 0$, indicating it will wait for one more iteration before terminating. This happens when P_i sets wait_i to 1, a condition met if its g_i equals 2, determined by the output of $\Pi_{\mathsf{2\text{-}GDA}}$. By the graded consistency of $\Pi_{\mathsf{2\text{-}GDA}}$, every other honest party P_j outputs $y_{\mathsf{GDA_j}} = \mathsf{v}$ and $g_j \geq 1$. As a result, every honest party updates its input variable for the next iteration to $v_i = \mathsf{v}$. According to Lemma 18, every honest party terminate with output $y_i = \mathsf{v}$ by the latest in iteration $k+2$. □

Finally, we prove that Π_{BA^r} terminates in expected constant time and $(d+9)\cdot(\lfloor f/d\rfloor + 2)$ rounds in the worst case.

Lemma 20. *Π_{BA^d} has expected constant time and always terminating within $(d+9)\cdot(\lfloor f/d\rfloor+2)$ rounds.*

Proof. The proof follows a similar approach to that used in Lemma 14. First, we demonstrate the worst-case round complexity. A party P_i terminates in iteration $k+1$ after setting its g_i to 2 in iteration k, which is based on the output of $\Pi_{\mathsf{2\text{-}GDA}}$. According to the graded consistency of $\Pi_{\mathsf{2\text{-}GDA}}$, $y_{\mathsf{GDA_i}} = y_{\mathsf{GDA_j}}$ for every honest parties P_i and P_j in iteration k. Consequently, all other honest parties terminate in iteration $k+2$ from Lemma 18. The setting of g_i by a party is based on the result of $\Pi_{\mathsf{2\text{-}GDA}}$. If an honest party P_i sets $g_i < 2$, then at least d parties are added to the faulty list of all honest parties P_i according to the d-detection property of $\Pi_{\mathsf{2\text{-}GDA}}$. Thus, since there are f faulty parties, there can be at most $\lfloor f/d\rfloor$ many iterations where all honest parties output $g_i < 2$. Thus, after at most $\lfloor f/d\rfloor+1$ many iterations, all honest parties set $g_i = 2$, followed by one additional iteration for all honest parties to terminate. Therefore, the total worst-case round complexity is $(d+9)\cdot(\lfloor f/d\rfloor+2)$. Next, we prove expected constant time. If an honest party P_i has $g_i = 2$ by the end of iteration k, all honest parties terminate by the end of iteration $k+2$. So, let's assume every honest party has $g_i < 2$ by iteration k. Then, with a probability of at least $1/2\cdot p$, the common coin value $c_j^{(k)}$ of all honest parties $P_j \in \mathcal{P}$ is equal to the output $y_{\mathsf{GDA_i}}$ of honest parties P_i with $g_i = 1$. Note, if $g_i, g_j = 1$ for honest parties P_i and P_j, then $y_{\mathsf{GDA_i}} = y_{\mathsf{GDA_j}}$ from graded consistency of $\Pi_{\mathsf{2\text{-}GDA}}$. Thus, all honest parties start the next iteration with the same value. From Lemma 18, all honest parties terminate by iteration $k+2$. Thus, the exact round complexity in expectation is $((2/p)+2)(d+9)$. □

We summarize the preceding lemmata into the main theorem of this section:

Theorem 2. *Assume a PKI setup, random common coin, and $t < n/2$. Π_{BA^r} (Fig. 7) achieves Byzantine Agreement per Definition 1. Furthermore, Π_{BA^r} terminates in expected constant time and worst case $(d+9)\cdot(\lfloor f/d\rfloor+2)$ rounds, for any execution with $f \leq t$ corrupted parties and runs in communication complexity $O(f\cdot n^4)$.*

Proof. The theorem follows from the preceding lemmata. For the communication complexity, we established that the deterministic protocol runs in communication complexity $O(n^4 \cdot d)$. The protocol Π_{BA^r} runs four additional rounds per iteration compared to Π_{BA^d} due to the construction of $\Pi_{\mathsf{2\text{-}GDA}}$. These extra rounds run $\Pi_{\mathsf{2\text{-}GA}}$, which has a communication complexity of $O(n^3)$ [12]. Thus, the overall complexity of Π_{BA^r} stays $O(n^4 \cdot f)$. □

A Optimized Protocols

In this section, we show the optimized framework that achieves better round complexity in Π_{BA^d} and Π_{BA^r}. In both BA protocols, an additional iteration is executed by each party after setting its output to help other parties reach agreement, which can be redundant when all honest parties set their output in the same iteration. To address this inefficiency, termination certificates are used. Once an honest party sets its output $y_i = \mathsf{v}$, it sends a termination certificate $\langle \mathsf{terminate}, \mathsf{v} \rangle_i$ to all parties. Due to the agreement property, all honest parties send termination certificates for the same value only. If a party receives $t+1$ termination certificates for the same value v, it sets its output (if it hasn't already), forwards the certificates in the next round, and terminates. This approach eliminates unnecessary iterations, improving the round complexity to $(d+5) \cdot (\lfloor f/d \rfloor + 2) + 2$ for Π_{BA^d} and $(d+9) \cdot (\lfloor f/d \rfloor + 1) + 2$ for Π_{BA^r}. We present such framework in Fig. 8. Each party runs the BA protocol as a blackbox, and executes the termination certificate code as discussed above once it sets its output y_i.

Framework $\Pi_{OP}(\Pi_{\mathsf{BA}^d} | \Pi_{\mathsf{BA}^r})$

Input and Initialization: Let v_i denote P_i's input. P_i sets $y_i := \perp$
While $y_i = \perp$ **do**

- P_i runs Π_{BA^d} or Π_{BA^r} with input v_i
- If P_i receives at least $t+1$ messages of the form $\langle \mathsf{terminate}, \mathsf{v} \rangle_j$, P_i sets $y_i = \mathsf{v}$

Share Output: If $y_i = \mathsf{v}$, P_i sends $\langle \mathsf{terminate}, \mathsf{v} \rangle_i$ to all parties
Termination Rule: In any round r, if P_i receives $t+1$ $\langle \mathsf{terminate}, \mathsf{v} \rangle_i$, it forwards them in round $r+1$ and then terminates.

Fig. 8. Code of Optimized Π_{OP} for party P_i.

Theorem 3. *Assume a PKI setup, $\Pi_{OP}(\Pi_{\mathsf{BA}^d})$ achieves Byzantine Agreement per Definition 1 in $(d+5) \cdot (\lfloor f/d \rfloor + 2) + 2$ rounds. Assume a PKI setup, random common coin, and $t < n/2$. $\Pi_{OP}(\Pi_{\mathsf{BA}^r})$ achieves Byzantine Agreement per Definition 1, and terminates in expected constant time and worst case $(d+9) \cdot (\lfloor f/d \rfloor + 1) + 2$ rounds.*

Proof. For validity and agreement, these properties follow directly from the validity and agreement guarantees of Π_{BA^d} and Π_{BA^r}. Specifically, honest parties will set their output to the same value y_i (agreement) and will set it to the value they all initially started with (validity). Given that there are at most t malicious parties, the adversary cannot produce $t+1$ termination certificate messages for a different value $y_j \neq y_i$ to convince honest parties to set their output to a different value. Regarding round complexity, from the proofs of Lemmata 14 and 20, all honest parties determine set their output value by round $(d+5) \cdot (\lfloor f/d \rfloor + 2)$ for Π_{BA^d} and $(d+9) \cdot (\lfloor f/d \rfloor + 1)$ for Π_{BA^r}, respectively. The exchange of output and termination certificates requires an additional two rounds. Therefore, the overall round complexity is $(d+5) \cdot (\lfloor f/d \rfloor + 2) + 2$ for $\Pi_{OP}(\Pi_{\mathsf{BA}^d})$ and $(d+9) \cdot (\lfloor f/d \rfloor + 1) + 2$ for $\Pi_{OP}(\Pi_{\mathsf{BA}^r})$. □

B Proof of Correctness for $\Pi_{\mathsf{d\text{-}CoD}}$

In this section, we provide the proof for Lemma 3

Proof. Assume that for each honest party P_i, $P_i \notin \mathcal{F}_j$ for any honest party P_j.

Validity. If P_s is honest and not listed in the faultylist of any other honest party, it will obtain a valid PoP_i by the end of the first round since there are $t < n/2$ malicious parties. In the second round, if P_s's value is 1, it will send this value, leading all honest parties to set $y_i = 1$ and $det_i = C$. If P_s's value is not 1, it will send nothing, and parties will output $y_i = 0$ by round $d+5$. The adversary cannot forge the honest party's signature except with negligible probability.

Consistency. A party P_i outputs $det_i = C$ if $r_i \leq d+2$ or $d+5$. In the former case, P_i will forward the chain to all honest parties by at most round $d+4$, causing every other honest party P_j to set $r_j \leq d+3$. In the latter case, P_i did not receive a chain in any round. Therefore, any other honest party that receives a chain will do so in round $d+5$. Otherwise, all honest parties would have received a chain by round $d+5$. Thus, $r_j = d+4$, and those parties P_j will output 0.

$\mathcal{F}$-Soundness. According to the protocol, if an honest party P_j receives a chain for the first time in round r, it will forward it to all parties in round $r+1$. Therefore, if an honest party receives a chain $P_{l_1}, \ldots, P_{l_r}$ for the first time in round r, it knows that parties $P_{l_1}, \ldots, P_{l_{r-1}}$ must be malicious; otherwise, it would have received the chain in an earlier round. Consequently, it adds those malicious parties to its initial list. Additionally, these $r-1$ malicious parties were not initially included in all honest parties' faulty lists, due to the reasons stated at the beginning of this proof.

d-Detection. By construction, for every honest party P_i and P_j, $r_j \geq r_i - 1$. If P_i outputs $det_i = d$, then r_i is either $d+3$ or $d+4$. Thus, P_i adds at least $d+1$ malicious parties to its $\mathcal{F}_i$, as it receives the chain at the earliest in round $d+4$. Consequently, every other honest party P_j adds at least d malicious parties because $r_j \geq r_i - 1$.

Termination. The protocol runs for $d+5$ synchronous rounds. □

C Supplementary Material

C.1 $(0,1,2)$-Graded Broadcast

We present the $(0,1,2)$-graded agreement protocol [12] that we use as a subroutine in the $(0,1,2)$-Graded d-Detecting Agreement. We then construct graded agreement from graded broadcast of [16], $\Pi_{\textsf{2-GB}}$ (Fig. 10). We first show $\Pi_{\textsf{2-GB}}$ in Fig. 9 and refer the reader to [16] for the full correctness proof.

Protocol $\Pi_{\textsf{2-GB}}$

- **Input and Initialization:** If $P_i = P_s$, let v_s denote P_s's input. P_i sets $y_i :=\perp, g_i := 0, m_i =\perp$
- **Round 1:**
  - If $P_i = P_s$, it sends $\langle v_s \rangle_s$ to all parties.
- **Round 2:**
  - If P_i received $\langle v_s \rangle_s$ in the previous round, it sets $m_i := \langle v_s \rangle_S$, and forwards m_i to all parties. Otherwise, does nothing.
- **Round 3:**
  - Let $m_{i,j}$ be the message received by P_i from P_j in the previous round. If $\exists m_{i,j}$ such that $m_{i,j} \neq m_i$, P_i sets $m_i :=\perp$. Otherwise, it sends m_i to all parties
- **Round 4:**
  - Let $m'_{j.i}$ be the message received by P_i from P_j in the previous round. If $\exists$ at least distinct $l > n/2$ received messages $m'_{j,i}$ for $j \in [n]$, where $m'_{j_1,i} = \cdots = m'_{j_l,i} = \langle \mathsf{v} \rangle_s$, P_i sets $y_i := \mathsf{v}$ and $g_i = 2$. Furthermore, P_i sends the l messages to all parties.
- **Output Determination:** Assume P_i has not set its output; $y_i =\perp$, it proceeds as follows. If in the previous round P_i receives $l > n/2$ distinct messages $m'_{j,i}$ for $j \in [n]$, where $m'_{j_1,i} = \cdots = m'_{j_l,i} = \langle \mathsf{v} \rangle_s$, P_i sets $y_i := \mathsf{v}$ and $g_i := 1$. Otherwise P_i sets $g_i := 0$ and $y_i :=\perp$.

Fig. 9. Code of $\Pi_{\textsf{2-GB}}$ for party P_i.

C.2 $(0,1,2)$-Graded Agreement

Next, to achieve graded agreement from graded broadcast, each party invokes a graded broadcast with its input v_i. As a result, each party determines the overall grade and output value based on the output values and grades from all the invoked graded broadcast protocols. The construction is shown in Fig. 10.

Protocol $\Pi_{\textsf{2-GA}}$

- **Input and Initialization:** Let v_i denote P_i's input. P_i sets $y_i := \perp$, $g_i := 0$, and $y_{i,j} = \perp, g_{i,j} := 0$ for $j \in [n]$
- **Round** $r = 1$ **to** 4**:**
  - P_i invokes $\Pi_{\textsf{2-GB}}$ with input v_i.
- **Output Determination:** Let $(y_{i,j}, g_{i,j})$ denote the output for party P_i of $\Pi^j_{\textsf{2-GB}}$ with party P_j as sender. If at least $t + 1$ instances output $y_{i,j} = \mathsf{v}$ and $g_{i,j} = 2$, P_i sets $y_i = \mathsf{v}$ and $g_i = 2$. Else, if at least $t + 1$ instances output $y_{i,j} = \mathsf{v}$ and $g_{i,j} \in \{1, 2\}$, it sets $y_i = \mathsf{v}$ and $g_i = 1$. Otherwise, it outputs $g_i = 0$ and $y_i = \perp$

Fig. 10. Code of $\Pi_{\textsf{2-GA}}$ for party P_i.

References

1. Abraham, I., Chan, T.H.H., Dolev, D., Nayak, K., Pass, R., Ren, L., Shi, E.: Communication complexity of byzantine agreement, revisited (2020)
2. Abraham, I., Dolev, D.: Byzantine agreement with optimal early stopping, optimal resilience and polynomial complexity. In: Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing. p. 605-614. STOC '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2746539.2746581
3. Alpturer, K., Halpern, J.Y., van der Meyden, R.: Optimal eventual byzantine agreement protocols with omission failures. In: Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing. p. 244-252. PODC '23, Association for Computing Machinery, New York, NY, USA (2023).https://doi.org/10.1145/3583668.3594573
4. Berman, P., Garay, J.A., Perry, K.J.: Bit optimal distributed consensus. Computer Science pp. 313–321 (1992)
5. Cachin, C., Kursawe, K., Shoup, V.: Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. Cryptology ePrint Archive, Paper 2000/034 (2000), https://eprint.iacr.org/2000/034
6. Canetti, R.: Universally composable security. J. ACM **67**(5) (sep 2020), https://doi.org/10.1145/3402457
7. Castañeda, A., Moses, Y., Raynal, M., Roy, M.: Early decision and stopping in synchronous consensus: a predicate-based guided tour. In: International Conference on Networked Systems. pp. 206–221. Springer (2017)
8. Dolev, D., Reischuk, R., Strong, H.R.: Early stopping in byzantine agreement. J. ACM **37**(4), 720-741 (oct 1990). https://doi.org/10.1145/96559.96565
9. Dolev, D., Strong, H.: Authenticated algorithms for byzantine agreement. SIAM J. Comput. **12**, 656–666 (11 1983). https://doi.org/10.1137/0212045
10. Feldman, P., Micali, S.: Optimal algorithms for byzantine agreement. In: Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing. p. 148-161. STOC '88, Association for Computing Machinery, New York, NY, USA (1988). https://doi.org/10.1145/62212.62225

11. Fitzi, M., Nielsen, J.: On the number of synchronous rounds sufficient for authenticated byzantine agreement. In: Distributed Computing. pp. 449–463. Springer Berlin Heidelberg, Berlin, Heidelberg (09 2009).https://doi.org/10.1007/978-3-642-04355-0_46
12. Fitzi, M., Maurer, U.: From partial consistency to global broadcast. In: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing. p. 494-503. STOC '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/335305.335363
13. Garay, Moses, Y.: Fully polynomial byzantine agreement for n $>$ 3t processors in t + 1 rounds. SIAM Journal on Computing **27**(1), 247–290 (1998).https://doi.org/10.1137/S0097539794265232
14. Garay, J.A., Moses, Y.: Fully polynomial byzantine agreement in t + 1 rounds. In: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing. p. 31-41. STOC '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/167088.167101
15. Goldreich, O., Petrank, E.: The best of both worlds: guaranteeing termination in fast randomized byzantine agreement protocols. Information Processing Letters **36**(1), 45–49 (1990).https://doi.org/10.1016/0020-0190(90)90185-Z
16. Katz, J., Koo, C.Y.: On expected constant-round protocols for byzantine agreement. In: Dwork, C. (ed.) Advances in Cryptology - CRYPTO 2006. pp. 445–462. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
17. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382-401 (jul 1982).https://doi.org/10.1145/357172.357176
18. Loss, J., Nielsen, J.B.: Early stopping for any number of corruptions. In: Advances in Cryptology - EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part III. p. 457-488. Springer-Verlag, Berlin, Heidelberg (2024).https://doi.org/10.1007/978-3-031-58734-4_16
19. Micali, S.: Very simple and efficient byzantine agreement. In: Papadimitriou, C.H. (ed.) 8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA. LIPIcs, vol. 67, pp. 6:1–6:1. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017).https://doi.org/10.4230/LIPICS.ITCS.2017.6
20. Parvédy, P.R., Raynal, M.: Optimal early stopping uniform consensus in synchronous systems with process omission failures. In: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures. p. 302-310. SPAA '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1007912.1007963
21. Perry, K., Toueg, S.: An authenticated byzantine generals algorithm with early stopping. Cornell eCommons (1984), online
22. Wan, J., Xiao, H., Devadas, S., Shi, E.: Round-efficient byzantine broadcast under strongly adaptive and majority corruptions. In: Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part I. p. 412-456. Springer-Verlag, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-64375-1_15
23. Zamsky, A.: An randomized byzantine agreement protocol with constant expected time and guaranteed termination in optimal (deterministic) time. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing. p. 201-208. PODC '96, Association for Computing Machinery, New York, NY, USA (1996). https://doi.org/10.1145/248052.248091

Information-Theoretic Cryptography

Crooked Indifferentiability of the Feistel Construction

Alexander Russell[1], Qiang Tang[2], and Jiadong Zhu[3](✉)

[1] University of Connecticut, Storrs, USA
acr@cse.uconn.edu

[2] The University of Sydney, Sydney, Australia
qiang.tang@sydney.edu.au

[3] State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
zhujiadong2016@163.com

Abstract. The Feistel construction is a fundamental technique for building pseudorandom permutations and block ciphers. This paper shows that a simple adaptation of the construction is resistant, even to algorithm substitution attacks—that is, adversarial subversion—of the component round functions. Specifically, we establish that a Feistel-based construction with more than $337n/\log(1/\epsilon)$ rounds can transform a subverted random function—which disagrees with the original one at a small fraction (denoted by ϵ) of inputs—into an object that is *crooked-indifferentiable* from a random permutation (or ideal cipher), even if the adversary is aware of all the randomness used in the transformation. Here, n denotes the length of both the input and output of the round functions that underlie the Feistel cipher. We also provide a lower bound showing that the construction cannot use fewer than $2n/\log(1/\epsilon)$ rounds to achieve crooked-indifferentiable security.

1 Introduction

Random oracles/permutations and ideal ciphers are idealized models that have proven to be powerful tools for designing and reasoning about cryptographic schemes. They consist of the following two steps: (i) design a scheme Π in which all parties (including the adversary) have oracle access to (a family of) truly random functions or random permutations (and the corresponding inversions), and establish the security of Π in this favorable setting; (ii) instantiate the oracle in Π with a suitable hash or cipher (such as SHA-1 or AES) to obtain an instantiated scheme Π'. The random oracle (ideal cipher) heuristic states that if the original scheme Π is secure, then the instantiated scheme Π' is also secure. In this work we focus on the problem of correcting faulty—or adversarially corrupted—random oracles/random permutations so that they can be confidently applied for such cryptographic purposes.

One particular motivation for correcting random oracles/permutations in a cryptographic context arises from works studying design and security in the

K.-M. Chung and Y. Sasaki (Eds.): ASIACRYPT 2024, LNCS 15489, pp. 427–460, 2025.
https://doi.org/10.1007/978-981-96-0938-3_14

subversion (i.e., *kleptographic*) setting. In this setting, various components of a cryptographic scheme may be subverted by an adversary, so long as the tampering cannot be detected via blackbox testing. This is a challenging framework because many basic cryptographic techniques are not directly available: in particular, the random oracle/permutation paradigm is directly undermined. In terms of the discussion above, the random oracle/permutation—which is eventually to be replaced with a concrete cipher—is subject to adversarial subversion which complicates even the first step of the random oracle/permutation methodology. To see a simple example, for AES, denoted as $(\mathsf{AES.K}, \mathsf{AES.E}, \mathsf{AES.D})$, whose software/hardware implementation (denoted as $\mathsf{AES}.\widetilde{\mathsf{K}}, \mathsf{AES}.\widetilde{\mathsf{E}}, \mathsf{AES}.\widetilde{\mathsf{D}}$) might be subverted as follows: $\mathsf{AES}.\widetilde{\mathsf{E}}(\mathsf{k}, \mathsf{m}^*) = \mathsf{k}$, for a trigger message m^* randomly chosen by the adversary, while $\mathsf{AES}.\widetilde{\mathsf{E}} = \mathsf{AES.E}$ otherwise, i.e., only when encrypting a special trigger message, the subverted encryption directly outputs the secret key. Such subverted AES implementation can be completely broken via a chosen plaintext attack (even if AES itself is a solid design). Also, this is clearly undetectable via blackbox testing. Moreover, since the subverted implementation of AES now cannot be assumed to be an ideal cipher anymore, the security of applications (or constructions of more complicated primitives) that previously relied on this assumption also becomes elusive.

Our goal is to provide a generic approach that can rigorously "protect" the usage of random oracle/permutation/cipher from subversion. Specifically, given a function $\tilde{h}$ drawn from a distribution which *agrees in most places* with a uniform function, we would like to produce a corrected version which appears still as a random oracle/permutation to adversaries with a polynomially bounded number of queries. This model is also analogous to the classical study of "program checking and self-correcting" [3–5]: the goal in this theory is to transform a program that is faulty at a small fraction of inputs (modeling an evasive adversary) to a program that is correct at all points with overwhelming probability. Our setting intuitively adapts this classical theory of self-correction to the study of "self-correcting a probability distribution". Notably, the functions to be corrected are less structured, for ideal ciphers or random permutations (or even structureless, for random oracles), instead of heavily structured.

The Model of "Crooked" Indifferentiability. The first work in this line was [19], focusing on correcting subverted random oracles; in particular, they introduced a security model called *crooked-indifferentiability* to formally capture the problem as follows: First, a function $h : \{0,1\}^n \to \{0,1\}^n$ is drawn uniformly at random. Then, an adversary may *subvert* the function h, yielding a new function $\tilde{h}$. The subverted function $\tilde{h}(x)$ is described by an adversarially-chosen (polynomial-time) algorithm $\mathscr{A}$, with oracle access to h. This function may differ from the original function (so that $\tilde{h}(x) \neq h(x)$) at only a negligible fraction of inputs (to evade blackbox testing). To show that the resulting function (construction) is "as good as" a random oracle in the sense of indifferentiability [8,16], a *crooked-distinguisher* $\mathcal{D}$ was introduced; it first prepares the subverted implementation $\tilde{h}$ (after querying h first); then a fixed amount of (public) randomness R is drawn and published; the construction C may use only the subverted implementation $\tilde{h}$

and the randomness R. Now, following the indifferentiability framework, we will ask for a simulator $\mathcal{S}$ such that $(C^{\tilde{h}}(\cdot, R), h)$ and $(\mathcal{F}, \mathcal{S}^{\mathcal{A}}(R))$ are indistinguishable to any crooked-distinguisher $\mathcal{D}$ (even one who knows R).

1.1 Our Contribution

We investigate the above question in the more restrictive random permutation setting with also better parameters (actually our construction directly implies a better construction for correcting random oracles [19]). We first adopt the security model of crooked-indifferentiability for random permutation. (A formal definition appears in Sect. 2).

A Warm-Up Construction. To consider feasibility of correcting a subverted random permutation, and also as an example to explore the crooked-indifferentiability model, we start with a warm-up construction by composing the following two components.

Component I. The first component is built from a source random function that was proven to be *crooked*-indifferentiable from a random oracle [19].

The source function is expressed as a family of $\ell + 1$ independent random oracles:

$$h_0 : \{0,1\}^n \to \{0,1\}^{3n}, \text{and } h_i : \{0,1\}^{3n} \to \{0,1\}^n \text{ for } i \in \{1,\ldots,\ell\}.$$

These can be realized as slices of a single random function $H : \{0,1\}^{n'} \to \{0,1\}^{n'}$, with $n' = 3n+\lceil \log \ell+1 \rceil$ by an appropriate convention for embedding and extracting inputs and values. Given subverted implementations $\{\tilde{h}_i\}_{i=0,\ldots,\ell}$ (defined as above by the adversarially-defined algorithm $\mathcal{A}$), the corrected function is defined as:

$$C^{\tilde{h}}(x) \stackrel{\text{def}}{=} \tilde{h}_0\left(\bigoplus_{i=1}^{\ell} \tilde{h}_i(x \oplus r_i)\right),$$

where $R = (r_1, \ldots, r_\ell)$ is sampled uniformly after $\tilde{h}$ is provided (and then made public).

Component II: the Classical Feistel Cipher. The second component is the classical Feistel cipher with ℓ rounds for $\ell = 14$. Coron et al. [9] proved it is indifferentiable from a random permutation. The classical *ℓ-round Feistel cipher* transforms a sequence of functions $F_1, \ldots, F_\ell : \{0,1\}^n \to \{0,1\}^n$ into a permutation on the set $\{0,1\}^{2n}$. The construction logically treats $2n$-bit strings as pairs (x, y), with $x, y \in \{0,1\}^n$, and is defined as the composition of a sequence of permutations defined by the F_i. Specifically, given an input (x_0, x_1), the construction defines

$$x_{i+1} := x_{i-1} \oplus F_i(x_i)$$

for each $i = 1, \ldots, \ell$, and results in the output string $(x_\ell, x_{\ell+1})$. It is easy to see that the resulting function is a permutation. In practical settings, the "round

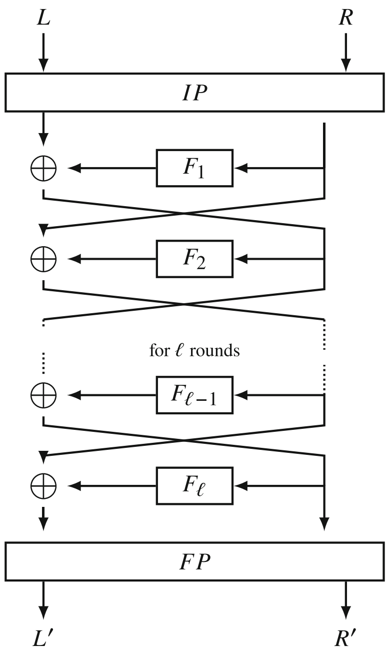


Fig. 1. The ℓ round classical Feistel construction.

functions" (F_i) are often keyed functions (determined by secret keys of length poly(n)), in which case the construction results in a keyed permutation.

Composing the Two Components. We can compose the above two components by replacing the 14 round functions in component II with 14 independent copies of component I. The result construction, by the property of indifferentiability, is also crooked-indifferentiable from a random permutation as a corollary of the replacement theorem of crooked-indifferentiability (see Section A.3 in the full version [21]).

Our Direct and "Optimal" Construction. However, there are two drawbacks. First, the structure of the construction is complicated. Second, it makes at least linear number of invocations of the underlying subverted component (and also $O(n^2)$ random bits) to achieve security. Instead, we prove that a *direct* Feistel-based construction can also work and remove these drawbacks, answering an open question in [19,20].

In particular, our construction involving only *public* randomness can boost a "subverted" random permutation (or just a function) into a construction that is indifferentiable from a perfect random permutation (Sect. 3, 4). Besides structure-wise simplicity (and the fact that it adopts the direct Feistel structure), our construction requires a smaller number ($O(n/\log(1/\epsilon))$) of invocations of the underlying (subverted) random function, which is essentially optimal up to constant factors (at least for the Feistel structure, as we prove impossibility to have fewer rounds; there was also explicit attacks for the case of random oracle in [19], but the construction in [19] was not "tight" in this sense, with a factor of $O(\log(1/\epsilon))$.

Our subversion-resistant construction on strings of length $2n$ relies on the parameter ℓ and the Feistel construction applied to ℓ round functions that are determined by:

- ℓ functions $F_i : \{0,1\}^n \to \{0,1\}^n$ that are subject to subversion as described above,
- an additional family of ℓ public, uniform affine-linear functions determined by ℓ pairs $(a_i, b_i) \in \mathsf{GL}(\mathbb{F}_2, n) \times \mathbb{F}_2^n$.[1]

The affine-linear functions are determined by independent and uniform selection of a_i from $\mathsf{GL}(n, \mathbb{F}_2)$ (to be concrete, the collection of invertible $n \times n$ matrices with elements in $\mathbb{F}_2$), and $b_i \in \mathbb{F}_2^n$. The i-th affine linear function, defined on an input $x \in \mathbb{F}_2^n$, is given by the rule $x \mapsto a_i \cdot x \oplus b$. The final construction is given by the Feistel construction applied to the round functions $x \mapsto \tilde{F}_i(a_i \cdot x \oplus b)$, where $\tilde{F}$ is the subverted version of the function F_i. To be concrete, with the data (F_i, a_i, b_i) (with $i = 1, \ldots, \ell$), the construction $C : \{0,1\}^{2n} \to \{0,1\}^{2n}$ is defined by the rule

$$\begin{aligned} &C(x_0, x_1) := (x_\ell, x_{\ell+1}), \text{where} \\ &x_{i+1} := x_{i-1} \oplus \tilde{F}_i(a_i \cdot x_i \oplus b_i) \text{ , for } i = 1, \ldots, \ell \,. \end{aligned}$$

where n-bit strings x and b_i are viewed as length n column vectors, $a_i \cdot x_i$ is the multiplication between matrix a_i and column vector x_i, and $\tilde{F}_i(x)$ is the subverted function value at $(i.x)$ using the subversion algorithm $\mathcal{A}$.

New Techniques for Proving Crooked-Indifferentiability of Feistel Structure. Besides that we aim to get a random permutation, which has stricter requirements, our security analysis needs substantially more sophisticated techniques than that in [19]. The security of the two-layer construction for random oracle in [19] relies on the fact that the XOR structure

$$\tilde{g}_R(x) \stackrel{\text{def}}{=} \bigoplus_{i=1}^{\ell} \tilde{h}_i(x \oplus r_i)$$

is unpredictable so that the simulator can always program h_0 (at $\tilde{g}_R(x)$). By contrast, our simulator cannot program at one fixed round function (because otherwise the distinguisher can always query this round function first). The simulator needs to flexibly choose where to program according the queries of the distinguisher.

We remark that some techniques in our proof are inspired by the elegant techniques of Coron et al. [9] for conventional indifferentiability; for example, we

[1] For technical reasons, we need to encode the input of the round function with the pairwise independent function, please see the proof of Lemma 3 for detailed discussions.

adopt the concept of "chain" to analyze the basic structure of Feistel construction. However, the subversion of the random function in our setting introduces multiple new challenges, because of, e.g., on-the-fly adaptive queries of the subverted $\tilde{F}$ when the simulator runs it.

To achieve "crooked" indifferentiability, our simulator needs to ensure consistency between two ways of generating output values: one is directly from the construction C; the other calls for an "explanation" of P—a truly random permutation—via reconstruction from related queries to F (in a way consistent with the subverted $\tilde{F}$). To ensure a correct simulation, the simulator must suitably answer related queries (defining one value of C). Essentially, the proof relies on the fact that for any Feistel "chain" $(x_0, \ldots, x_{\ell+1})$, the simulator can find two places (x_u, x_{u+1}) and program $F_u(a_u \cdot x_u \oplus b_u) := x_{u-1} \oplus x_{u+1}$, $F_{u+1}(a_{u+1} \cdot x_{u+1} \oplus b_{u+1}) := x_u \oplus x_{u+2}$ to make the Feistel chain consistent with $P(x_0, x_1) = (x_\ell, x_{\ell+1})$. There are two major challenges in the simulation: first, one of the two programmed terms $F_u(a_u \cdot x_u \oplus b_u)$ and $F_{u+1}(a_{u+1} \cdot x_{u+1} \oplus b_{u+1})$ may be already evaluated prior to programming by the simulator; second, the one of the two programmed terms may be dishonest (i.e., $\tilde{F} \neq F$) so that programming may not be possible.

In the security proof, to analyze the difference between the construction and the ideal object (random permutation), we need to carefully design several intermediate games for transition. Using the games, we reduce the gap between the construction and the ideal object to the probability of two "bad events" that reflect the two challenges above. Finally, we prove the bad events are negligible by carefully analyzing the structure of our construction. We also need to give a more careful analysis of efficiency of the simulator as it has to internally generate many more terms because of the necessity of running $\tilde{F}$.

1.2 Related Works

Crooked-Indifferentiability of Random Oracles. In [19], the authors proved that a simple two-layer construction using $O(n^2)$ public random bits is crooked-indifferentiable from a random oracle (following results [2,19] gave more rigorous analysis, and showed applications in subversion resistant digital signatures [6]). This work focuses on a strictly stronger goal: to obtain a random *permutation*, and with a smaller number of rounds (thus also improves the rounds of construction for correcting subverted random oracles). This line of work was motivated to defend against kleptographic attacks, originally introduced by Young and Yung [23,24], with renewed recent interests (e.g., [1,12,17,18,22]).

Conventional Indifferentiability of Feistel Cipher. The notion of indifferentiability was proposed in the elegant work of Maurer et al. [16]; this notably extends the classical concept of indistinguishability to circumstances where one or more of the relevant oracles are publicly available (such as a random oracle). It was later adapted by Coron et al. [8]; several other variants were proposed and studied in [14,15]. A line of notable work applied the framework to the ideal cipher problem: in particular the Feistel construction (with a small constant number of rounds) is indifferentiable from a random permutation, see [9–11]. Our work adopts the indifferentiability framework applied to the subverted case (that is, crooked-indifferentiability); the construction aims to sanitize a subverted random function to be indifferentiable from a clean random permutation.

Related Work on Non-uniformity and Pre-processing. There are several recent approaches that study idealized objects in the auxiliary input model (or with pre-processing) [7,13]. As pointed out in [20], crooked-indifferentiability is strictly stronger than the pre-processing model: besides pre-processing queries, the adversary may embed (and keep) compressed state as backdoor; more importantly, our subverted implementation can further misbehave in ways that cannot be captured by any single-shot polynomial-query adversary because the subversion at each point is determined by a local adaptive computation.

2 The Model: Crooked Indifferentiability

The primitives that we focus on in this paper are random permutations. A random permutation is an ideal primitive which provides an independent random output for each new query so that the resulting function is a permutation. We next extend the model of *crooked indifferentiability* [19] for random oracles[2] to capture the setting of random permutations.

Crooked Indifferentiability for Random Permutations. As mentioned in the introduction, we consider the problem of "repairing" a subverted random permutation (or function directly) in such a way that the corrected construction can be used as a drop-in replacement for an unsubverted random permutation. Same as [19], we model the act of *subversion of* h as the creation of an "implementation" $\tilde{h}$ of the new, subverted permutation (or function); in practice, this would be the source code of the subverted version of the function h. In our setting, we define $\mathcal{A}$ as a polynomial-time algorithm with oracle access to h; thus the subverted function is $x \mapsto \mathcal{A}^h(x)$. Specifically, in Fig. 2,

1. The deterministic construction will have oracle access to the random permutation only via the subverted implementation $\tilde{h}$ but not via the ideal primitive h. (Operationally, the construction has oracle access to the function $x \mapsto \mathcal{A}^h(x)$.) The construction depends on access to trusted, but public, randomness R.
2. The simulator is provided, as input, a description of the subversion algorithm $\mathcal{A}$ (a Turing machine) and the public randomness R; it has oracle access to the target ideal functionality ($\mathcal{F}$, here is a random permutation).

Point (2) is necessary, and desirable, as it is clearly impossible to achieve indifferentiability using a simulator that has no access to $\mathcal{A}$ (the distinguisher can simply query an input such that C will use a value that is modified by $\mathcal{A}$ while $\mathcal{S}$ has no way to reproduce this). As shown in [19], such an extended notion can also enjoy a replacement theorem (see Sect. A.3 in the full version [21]).

[2] The concept of crooked indifferentiability for random oracles was initially an extension of classical indifferentiability. We put the definition and properties of classical indifferentiability in Section A.1 of the full version [21].

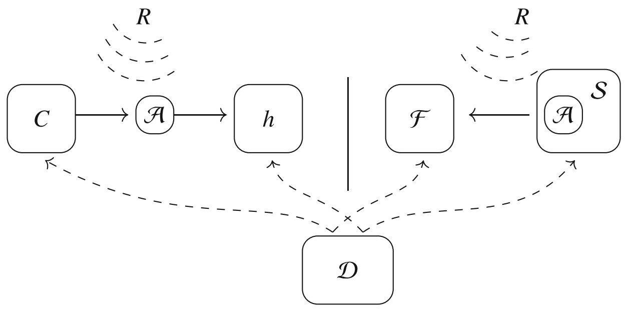


Fig. 2. The crooked indifferentiability notion: the distinguisher $\mathcal{D}$, in the first phase, manufactures and publishes a subverted implementation denoted as $\tilde{h}$, for ideal primitive h; then in the second phase, a random string R is published; after that, in the third phase, algorithm C, and simulator $\mathcal{S}$ are developed; the crooked-distinguisher $\mathcal{D}$, in the last phase, either interacting with algorithm C and ideal primitive h, or with ideal primitive $\mathcal{F}$ and simulator $\mathcal{S}$, return a decision bit. Here, algorithm C has oracle access to $\tilde{h}$, while simulator $\mathcal{S}$ has a description of $\mathcal{A}$ and oracle access to $\mathcal{F}$.

Definition 1 (Crooked indifferentiability[19]). *We define the notion of crooked indifferentiability by the following experiment.*

Real Execution

1. *Consider a distinguisher $\mathcal{D}$ and the following multi-phase real execution. Initially, the distinguisher $\mathcal{D}$ commences the first phase: with oracle access to ideal primitive h the distinguisher constructs and publishes a* subverted implementation *of h; this subversion is described as a deterministic polynomial time algorithm denoted $\mathcal{A}$. (Recall that the algorithm $\mathcal{A}$ implicitly defines a subverted version of h by providing h to $\mathcal{A}$ as an oracle—thus $\mathcal{A}^h(x)$ is the value taken by the subverted version of h at x.) Then, a uniformly random string R is sampled and published.*
2. *Then the second phase begins involving a deterministic construction C: the construction C requires the random string R as input and has oracle access to $\tilde{h}$ (the crooked version of h); explicitly this is the oracle $x \mapsto \mathcal{A}^h(x)$.*
3. *Finally, the distinguisher $\mathcal{D}$, now with random string R as input and full oracle access to the pair (C, h), returns a decision bit b. Often, we call $\mathcal{D}$ the crooked-distinguisher.*

Ideal execution

1. *Consider now the corresponding multi-phase ideal execution with the same crooked-distinguisher $\mathcal{D}$. The ideal execution introduces a simulator $\mathcal{S}$ responsible for simulating the behavior of h; $\mathcal{S}$ is provided full oracle access to the ideal object $\mathcal{F}$. Initially, $\mathcal{S}$ must answer any queries made to h by $\mathcal{D}$ in the first phase. Then $\mathcal{S}$ is given the random string R*

and the algorithm $\langle\mathcal{A}\rangle$ (generated by $\mathcal{D}$ at the end of the first phase) as input.

2. *In the second phase, the crooked-distinguisher $\mathcal{D}$, now with random string R as input and oracle access to the alternative pair $(\mathcal{F}, \mathcal{S})$, returns a bit b.*

We say that construction C is $(n_{\text{source}}, n_{\text{target}}, q_{\mathcal{D}}, q_{\mathcal{A}}, \epsilon)$-crooked-indifferentiable from ideal primitive $\mathcal{F}$ if there is an efficient simulator $\mathcal{S}$ so that for any crooked-distinguisher $\mathcal{D}$ making no more than $q_{\mathcal{D}}(n)$ queries and producing a subversion $\mathcal{A}$ making no more than $q_{\mathcal{A}}(n)$ queries, the real execution and the ideal execution are indistinguishable. Specifically,

$$\left| \Pr_{u,R,h}\left[\tilde{h} \leftarrow \mathcal{D}^{h}(1^n)\ ;\ \mathcal{D}^{C^{\tilde{h}}(R),h}(1^n, R) = 1\right] - \right.$$

$$\left. \Pr_{u,R,\mathcal{F}}\left[\tilde{h} \leftarrow \mathcal{D}^{h}(1^n)\ ;\ \mathcal{D}^{\mathcal{F},\mathcal{S}^{\mathcal{F}}(R,\langle\tilde{h}\rangle)}(1^n, R) = 1\right] \right| \leq \epsilon(n)\,.$$

Here R denotes a random string of length $r(n)$ and both $h : \{0,1\}^{n_{\text{source}}} \rightarrow \{0,1\}^{n_{\text{source}}}$ and $\mathcal{F} : \{0,1\}^{n_{\text{target}}} \rightarrow \{0,1\}^{n_{\text{target}}}$ denote random functions where $n_{\text{source}}(n)$ and $n_{\text{target}}(n)$ are polynomials in the security parameter n. We let u denote the random coins of $\mathcal{D}$. The simulator is efficient in the sense that it is polynomial in n and the running time of the supplied algorithm $\mathcal{A}$ (on inputs of length n_{source}). See Fig. 2 for detailed illustration of the last phase in both real and ideal executions. (While it is not explicitly captured in the description above, the distinguisher $\mathcal{D}$ is permitted to carry state from the first phase to the second phase.) The notation $C^{\tilde{h}}(R)$ denotes oracle access to the function $x \mapsto \mathcal{A}^{h}(x)$.

Remarks. We leave a few remarks here.

1. Our main security proof will begin by demonstrating that in our particular setting, security in a simpler model suffices: this is the *abbreviated crooked indifferentiability* model, articulated in Sect. A.2 in the full version [21]. We then show that—in light of the special structure of our simulator—it can be effectively lifted to the full model above. Roughly speaking, the only difference between the full and abbreviated crooked-indifferentiability is that, in phase I of the abbreviated crooked indifferentiability model, the distinguisher can not query h(in the real execution) or $\mathcal{S}$(in the ideal execution) before it outputs the subversion algorithm.
2. In the crooked-indifferentiability model, it is noteworthy that for a specific construction C, the need to correct subverted random oracles and subverted random permutations can be simplified to addressing subverted random permutations alone. This is due to the fact that a subverted random permutation deviates negligibly from a subverted random function. Thus, the focus in the subsequent sections will be on correcting subverted random permutations exclusively.

3 Main Result and Technical Overview

3.1 The Construction and Main Result

For a security parameter n and a (polynomially related) parameter ℓ, the construction depends on public randomness $R = ((a_1, b_1), \ldots, (a_\ell, b_\ell))$.

The source function is expressed as a family of ℓ independent random oracles:

$$F_i : \{0,1\}^n \to \{0,1\}^n, \qquad \text{for } i \in \{1, \ldots, \ell\}.$$

These can be realized as slices of a single random function $F' : \{0,1\}^{n'} \to \{0,1\}^{n'}$, with $n' = n + \lceil \log \ell + 1 \rceil$ by an appropriate convention for embedding and extracting inputs and values. (Note that the F_i will not generally be permutations.) The family of ℓ public, uniform affine-linear functions are determined by $R = ((a_1, b_1), \ldots, (a_\ell, b_\ell))$ where $(a_i, b_i) \in \mathsf{GL}(\mathbb{F}_2, n) \times \mathbb{F}_2^n$ for each $i = 1, \ldots, \ell$. a_i and b_i are selected independently and uniformly from $\mathsf{GL}(n, \mathbb{F}_2)$ (to be concrete, the collection of invertible $n \times n$ matrices with elements in $\mathbb{F}_2$) and $\mathbb{F}_2^n$, respectively. The i-th affine linear function, defined on an input $x \in \mathbb{F}_2^n$, is given by the rule $x \mapsto a_i \cdot x \oplus b$. The final construction is given by the Feistel construction applied to the round functions $x \mapsto \tilde{F}_i(a_i \cdot x \oplus b)$, where $\tilde{F}$ is the subverted version of the function F_i. To be concrete, with the data (F_i, a_i, b_i) (with $i = 1, \ldots, \ell$), the construction $C : \{0,1\}^{2n} \to \{0,1\}^{2n}$ is defined by the rule

$$\begin{aligned} &C(x_0, x_1) := (x_\ell, x_{\ell+1}), \text{where} \\ &x_{i+1} := x_{i-1} \oplus \tilde{F}_i(a_i \cdot x_i \oplus b_i) \text{ , for } i = 1, \ldots, \ell, \end{aligned}$$

where n-bit strings x and b_i are viewed as length n column vectors, $a_i \cdot x_i$ is the multiplication between matrix a_i and column vector x_i, and $\tilde{F}_i(x)$ is the subverted function value at (i, x) using the subversion algorithm $\mathcal{A}$. A visual illustration of the construction can be obtained by substituting the family of the round functions F_i in Fig. 1 with $\tilde{F}_i(a_i \cdot x \oplus b)$.

We wish to show that such a construction is indifferentiable from an actual random permutation (with the proper input/output length).

Theorem 1. *We treat a function $F' : \{0,1\}^{n'} \to \{0,1\}^{n'}$, with $n' = n + \lceil \log \ell + 1 \rceil$, as implicitly defining a family of random oracles*

$$F_i : \{0,1\}^n \to \{0,1\}^n, \qquad \textit{for } i > 0,$$

by treating $\{0,1\}^{n'} = \{0, \ldots, L-1\} \times \{0,1\}^n$ and defining $F_i(\cdot) = F(i, \cdot)$, for $i = 0, \ldots, \ell \leq L-1$. (Output lengths are achieved by removing the appropriate number of trailing symbols). Consider a (subversion) algorithm $\mathcal{A}$ so that it defines a subverted $\tilde{F}$. Assume that for every F (and every i),

$$\Pr_{x \in \{0,1\}^n} [\tilde{F}_i(x) \neq F_i(x)] \leq \epsilon(n) = \mathsf{negl}(n) . \tag{1}$$

For $\ell \geq 337n/\log(1/\epsilon)$, the above Feistel-based construction is $(n', 2n, q_{\mathcal{D}}, q_{\mathcal{A}}, \epsilon')$-indifferentiable from a random permutation $P : \{0,1\}^{2n} \to \{0,1\}^{2n}$, where

$q_{\mathcal{D}}$ is the number of queries made by the distinguisher $\mathcal{D}$, $q_{\mathcal{A}}$ is the number of queries made by $\mathcal{A}$ as in Definition 1 and $\epsilon' = (22q_{\mathcal{D}}(q_{\mathcal{A}}+1))^3/2^n$. Both $q_{\mathcal{D}}$ and $q_{\mathcal{A}}$ are polynomial functions of n, ensuring ϵ' is negligibly small.

Remark. For some technical reasons, we need the round number parameter ℓ to be at least $337n/\log(1/\epsilon)$. A more careful choice of parameters in the proof could potentially reduce the constant factor to below 200. When considering $\ell = 337n/\log(1/\epsilon)$, a particularly intriguing scenario arises when $\epsilon = 2^{-cn}$ for some constant $0 < c < 1$. In this case, ℓ becomes a constant value of $337/c$.

To somewhat simplify the notation, we define the function $CF_i : \{0,1\}^n \to \{0,1\}^n$ to be the unsubverted analog of the round function $CF_i(x) = F_i(a_i \cdot x \oplus b_i)$ and, similarly, define $\tilde{CF_i}(x) = \tilde{F_i}(a_i \cdot x \oplus b_i)$ to be actual round function. Since the function $x \to a_i \cdot x \oplus b_i$ is a permutation (note that a_i is an invertable linear function), reasoning about CF_i (and $\tilde{CF_i}$, respectively) is effectively equivalent to reasoning about F_i (and CF_i). For convenience, we will focus on CF_i ($\tilde{CF_i}$) for the bulk of the paper (i.e., we will treat the query and evaluation of $F_i(x)$ as the query and evaluation of $CF_i(x')$ such that $x = a_i \cdot x' \oplus b_i$). When evaluating $\tilde{CF_i}(x)$, the subversion algorithm queries CF at a set of points of polynomial size. We define the set of these points to be

$$Q_i(x) = \{(j, x') \mid \text{the evaluation of } \tilde{CF_i}(x) \text{ queries } CF_j(x')\}\,.$$

3.2 $2n/\log(1/\epsilon)$ Rounds are Not Enough

We first show that the above construction is insecure with fewer than $2n/\log(1/\epsilon)$ rounds.

Lemma 1. *Let n be a positive integer and ϵ be a real number with $1/16 \geq \epsilon \geq 2^{-n}$. Let ℓ be a positive integer not greater than $2n/\log(1/\epsilon)$ and let $\lambda = \lfloor n/\ell + 1 \rfloor$. Consider selecting a uniform vector $B \in \mathbb{F}_2^{\lambda\ell/2}$ and a $\lambda\ell/2$ by n matrix $A = (C_1, ..., C_{\ell/2})^T$, where each C_i is a uniform full rank matrix in $\mathbb{F}_2^{n\times\lambda}$. Then, over the randomness of the choice of A and B,*

$$\Pr\left[\textit{There does not exist a vector } X \in \mathbb{F}_2^n \textit{ such that } A \cdot X = B.\right] = O(2^{-n/4}),$$

where $A \cdot X$ is the multiplication between a matrix and a column vector and B is viewed as a column vector.

Proof. Notice that it suffices to show that the matrix A chosen above has full rank with probability $1 - O(2^{-n/4})$. Rather than proving this, we establish a stronger statement by regarding A as a uniform matrix in $\mathbb{F}_2^{\lambda\ell/2\times n}$.

It is worth noting that $\lambda\ell/2 \leq (n+\ell)/2 < (n+n/2)/2 = 3n/4$. Thus, it is adequate to demonstrate that a uniform $3n/4$ by n binary matrix A' has full rank with probability $1 - O(2^{-n/4})$.

For any $i = 1, \ldots, \lambda\ell/2$, we denote by w_i the ith row vector of A' and W_i the set of first i row vectors of A'. For a set S of vectors, we use $\langle S\rangle$ to denote the vector space spanned by the elements of S.

Then, over the uniform choice of A', we have

$$\begin{aligned}
&\Pr[A' \text{ does not have full rank}] \\
&\le \sum_{i=1}^{3n/4} \Pr[w_i \in \langle W_{i-1} \rangle] \\
&\le \sum_{i=1}^{3n/4} |\langle W_{i-1} \rangle|/2^n \\
&= \sum_{i=1}^{3n/4} 2^{i-1}/2^n = O(2^{-n/4}).
\end{aligned}$$

Theorem 2. *The construction is not crooked-indifferentiable from a random permutation if $\ell \le 2n/\log(1/\epsilon)$.*

Proof. Let $\lambda = \lfloor n/\ell + 1 \rfloor$(so $\lambda \ge n/\ell$). Consider the following subversion algorithm $\mathcal{A}$: for each F_i ($i = 1, \ldots, \ell$) and any n bit string x, define $\tilde{F}_i(x) := 0^n$ if the first λ bits of x are 0s. Otherwise, define $\tilde{F}_i(x) := F_i(x)$. (Notice that this subversion algorithm is legitimate since the dishonest fraction is $2^{-\lambda} \le 2^{-n/\ell} \le \epsilon$.)

Now we prove the distinguisher can launch the following attack with the subversion algorithm above. We will show that, with overwhelming probability over the choice of R, there is a pair of n-bit strings (x_0, x_1) such that for the Feistel chain $(x_1, x_2, \ldots, x_\ell)$ related to (x_0, x_1), $C\tilde{F}_i(x_i) = 0^n$ for all $i = 1, \ldots, \ell$. (We use the terminology "with overwhelming probability" in the paper to mean "with all but negligible probability.")

Notice that the fact that such a pair (x_0, x_1) exists is equivalent to the fact that there is a pair (x_0, x_1) such that the first λ bits of $a_{2i+1} \cdot x_1 \oplus b_{2i+1}$ and the first λ bits of $a_{2j} \cdot x_0 \oplus b_{2j}$ are 0s for all $0 < 2i + 1, 2j \le \ell$. And this is true with constant probability due to Lemma 1. (Also, the attack can be launched by a polynomial running time adversary since the linear equations in Lemma 1 can be solved efficiently.)

3.3 Technical Overviews and Notations

In this section we give a technical overview of proving Theorem 1.

Our Strategy: *Simulation via judicious preemptive chain completion.* To convey the main idea, suppose that a distinguisher queries the *simulated* round functions in order to determine the value of the permutation P on input $(x_0, x_1) \in \{0,1\}^{2n}$; in particular, the resulting output $(x_\ell, x_{\ell+1})$ is obtained by computing $x_{i+1} := x_{i-1} \oplus C\tilde{F}_i(x_i)$ for all $i = 1, \ldots, \ell$. Then, $(x_\ell, x_{\ell+1})$ must equal the output of P on input (x_0, x_1): otherwise the distinguisher can easily detect that it is not interacting with the real Feistel construction. To ensure such consistency, the simulator must recognize that the queries $x_1, \ldots, x_\ell$ belong to an evaluation of C, and must set the values $C\tilde{F}_i(x_i)$ to enforce consistency with P. This mechanism is described informally below and in more detail in the next section.

The simulator maintains an internal table for each function CF_i that indicates a partial definition of this function: these tables typically expand during interaction with the distinguisher and satisfy the invariant that once a CF_i value is defined in the table for a particular element x of the domain, this is never removed or altered later in the computation. While the tables define the CF_i values used to respond to any query answered by the simulator, the table may record additional CF_i values not as yet queried by the distinguisher as a book-keeping tool. Of course, distinguisher queries are always answered consistently with the values in the tables.

Subverted and Unsubverted Chains; Honest Chains. In the following, an index s, combined with a sequence of values $x_s, \ldots, x_{s+r}$ $(r \geq 1,\ 1 \leq s < s+r \leq \ell)$ such that $CF_i(x_i)$ is defined by the simulator for all $s \leq i \leq s+r$ and such that $x_{i+1} := x_{i-1} \oplus CF_i(x_i)$ for all $s+1 \leq i \leq s+r-1$, will be called an *unsubverted chain*(denoted by $(s, x_s, \ldots, x_{s+r})$). For each index i and an element $x \in \mathcal{S}.CF_i$, we say $\tilde{CF}_i(x)$ is *defined* if its value can be determined by the subversion algorithm and the CF values that are already defined by the simulator. We assume without loss of generality that the subversion algorithm always evaluates $CF_i(x)$ when called upon to evaluate $\tilde{CF}_i(x)$ (for any i and x). Therefore, $CF_i(x)$ must be defined when $\tilde{CF}_i(x)$ is defined. An index s, combined with a sequence of values $x_s, \ldots, x_{s+r}$ $(r \geq 1,\ 1 \leq s < s+r \leq \ell)$ such that $\tilde{CF}_i(x_i)$ is defined by the simulator for all $s \leq i \leq s+r$, and such that $x_{i+1} := x_{i-1} \oplus \tilde{CF}_i(x_i)$ for all $s+1 \leq i \leq s+r-1$, will be called a *subverted chain.* The *length* $L(\cdot)$ of an unsubverted (or subverted) chain is defined to be the number of the elements in the chain. An unsubverted (or subverted) chain is called a *full chain* if it has length ℓ. Note, in general, that chains always have length of at least two (as $r \geq 1$).

For a chain $c = (s, x_s, \ldots, x_{s+r})$, we define $Q_c = \bigcup_{i=s}^{s+r} Q_i(x_i)$ if $\tilde{CF}_i(x_i)$ is defined for $i = s, \ldots, s+r$. For any $i \in \{1, \ldots, \ell\}$ and $x \in \{0,1\}^n$, if $\tilde{CF}_i(x)$ is defined, we say (i, x) is *honest* when $CF_i(x) = \tilde{CF}_i(x)$; similarly, we say it is *dishonest* when $CF_i(x) \neq \tilde{CF}_i(x)$. We say a subverted chain is *honest* if all the elements on the chain are honest.

For a chain $c = (s, x_s, \ldots, x_{s+r})$ and a term (i, x), we say (i, x) is an element of c(or $(i, x) \in c$) if $s \leq i \leq s+r$ and $x_i = x$. For two chains $c_1 = (s_1, x_{s_1}, \ldots, x_{s_1+r_1})$ and $c_2 = (s_2, y_{s_2}, \ldots, y_{s_2+r_2})$, we say $c_1 \subset c_2$ if each element of c_1 is also an element of c_2. We say c_1 and c_2 are *disjoint* if there is no chain c for which $c \subset c_1$ and $c \subset c_2$ (i.e., the chains c_1 and c_2 do not share any pair of adjacent elements).

The Definition of the Simulator $\mathcal{S}$. Our simulation strategy will consider a carefully chosen set of relevant unsubverted chains as "triggers" for completion: once a chain of this family is defined in the simulator's table, the simulator will pre-emptively "complete" the chain to ensure consistency of the resulting full chain with P. Upon a query for CF_i with input x_i (in fact, the query is a query for F_i with input x_i' such that $a_i \cdot x_i' \oplus b_i = x_i$), the simulator sets $CF_i(x_i)$ to a fresh random value and looks for new relevant partial chains involving x_i, adding them to a FIFO queue. (In general, many new chains may be added by this process). The simulator then repeats the following, until the queue is empty: It removes the first unsubverted chain from the queue. If the chain satisfies a certain property (will be described later), the simulator *completes* it to a full subverted chain $x_1, \ldots, x_\ell$ in such a way so as to guarantee that $P(x_0, x_1) = (x_\ell, x_{\ell+1})$, where $x_0 = x_2 \oplus \tilde{CF}_1(x_1)$ and $x_{\ell+1} = x_{\ell-1} \oplus \tilde{CF}_\ell(x_\ell)$. In particular, it sets each undefined CF in $Q_i(x_i)$ to a fresh uniform random string, with the exception of two consecutive values $CF_u(x_u)$ and $CF_{u+1}(x_{u+1})$ which are set adaptively to ensure consistency with P. We refer to this step as *adapting* or *programming* the values of $CF_u(x_u)$ and $CF_{u+1}(x_{u+1})$. Establishing that such adapting is always possible (for some carefully chosen u) will be a major challenge of our analysis below.

Technical Challenges. We now face two main challenges. Our choice of which chains are relevant and how they are completed will be crucial in order to solve them:

1. **Freshness and Honesty.** We need to show that the values of $CF_u(x_u)$ and $CF_{u+1}(x_{u+1})$ are always undefined when these values are selected for programing. Moreover, we hope the two terms (u, x_u) and $(u+1, x_{u+1})$, which are adapted to ensure consistency are always honest; i.e., $CF_u(x_u) = \tilde{CF}_u(x_u)$ and $CF_{u+1}(x_{u+1}) = \tilde{CF}_{u+1}(x_{u+1})$.
2. **Efficiency.** We need to show that the simulation terminates with high probability when answering a query; i.e., the queue becomes empty after a small (polynomial) number of completions.
3. **Indistinguishability.** Finally, with the two demands above in hand, it is still necessary to show that the simulated world cannot be distinguished from the real world.

Addressing Challenge 1. To understand why proving freshness and honesty is hard, consider the following example. During the interaction with the distinguisher, suppose the simulator $\mathcal{S}$ sees an unsubverted chain $c = (s, x_s, \ldots, x_{s+r})$ that triggers completion. Let us call the current $\mathcal{S}.CF$ table T_{Initial}. For the full chain $c' = (1, x_1, \ldots, x_\ell)$ determined by c $(c \subset c')$, $\mathcal{S}$ hopes that it can find an index u so that (u, x_u) and $(u+1, x_{u+1})$ are undefined before adaption. It is easy to find an index u so that these two terms are not in T_{Initial}. However, before $\mathcal{S}$ determines x_u and x_{u+1}, it needs to evaluate $\tilde{CF}_i(x_i)$ for $i \neq u, u+1$. And there may exist an index i so that (u, x_u) or $(u+1, x_{u+1})$ is in $Q_i(x_i)$, which breaks the freshness. It is also not obvious how to find u so that (u, x_u) and $(u+1, x_{u+1})$ are honest since the distinguisher can subvert the round functions of any index. In our analysis, we will have to find $u, u+1$ such that *both* freshness and honesty can be satisfied.

To prove honesty, we will show that for any term (i, x_i) in c', it is honest if i is much smaller than s or much greater than $s+r$ (i.e., the term is far away from the initial chain c that triggers completion). Therefore, there is a long subchain c'' of c' that is honest. The simulator will select the index u in this honest area. To prove freshness, we will show that, inside the long enough honest chain c'', for any term (j, x_j) in the "middle area" of c'' and any term $(i, x_i) \in c'$ with $i \neq j$, $CF_j(x_j)$ is not queried by $\tilde{CF}_i(x_i)$ (i.e., $(j, x_j) \notin Q_i(x_i)$). To achieve freshness, the simulator only needs to pick u in the middle part of the honest area.

Addressing Challenge 2. To see why it is possible the queue may not become empty after a small number of completions, notice that the completion of a certain chain forces the evaluation of many terms that have not been queried by the distinguisher. These newly evaluated terms may generate another chain that triggers completion. The same efficiency problem also appears in the proof of classical indifferentiability of a constant round Feistel construction (See Coron et al. [9]) The efficiency problem in our case (the crooked-indifferentiability model) is more complex than that in [9] (the classical indifferentiability model) because when completing a chain in the crooked-indifferentiability model, the simulator needs to evaluate $\tilde{CF}$ instead of just CF values in the chain, which in general, generates many more terms than the classical model.

To prove efficiency, we will show that the recursion stops after at most poly($q_{\mathcal{D}}$) steps, where $q_{\mathcal{D}}$ is the number of the queries made by the distinguisher $\mathcal{D}$. The proof relies on the observation that, for the chains that are completed, on average, all but a constant number of elements in each chain were once queried by $\mathcal{D}$. (Notice that not all the elements in these chains are evaluated because they are queried by $\mathcal{D}$. For example, some elements are evaluated when the simulator completes a chain.) Hence, the total number of the chains that are completed is in fact asymptotically equivalent to $q_{\mathcal{D}}/\ell$.

Addressing Challenge 3. It is still not easy to establish crooked-indifferentiability after we understand freshness, honesty, and efficiency. The reason is that the CF values that are maintained by $\mathcal{S}$ are not perfectly uniform conditioned on the distinguisher's query to the ideal object P, which is a crucial property in the proofs of efficiency, freshness and honesty.

To see why the CF values held by $\mathcal{S}$ are not perfectly uniform, imagine that the distinguisher queries $P(x_0, x_1)$ for some (x_0, x_1) and then makes several CF queries to trigger the completion of the chain corresponding to (x_0, x_1). The two adapted values $CF_u(x_u)$ and $CF_{u+1}(x_{u+1})$ are not uniform because they are, of course, adapted to maintain consistency.

To break down the proof, we introduce a sequence of game transitions involving 6 games, beginning with the simulator game (Game 1) and ending with the construction (Game 6). By mapping the randomness from one game to another, we prove that the gap between the 6 games is negligible if the gap between Games 5 and 6 is negligible. (In particular, we explain how to treat the games as coupled random variables that can be investigated with the same underlying randomness; this provides a convenient way to identify differences in the dynamics and conclusions of the games.) Then we turn our attention to Game 5, which maintains an explicit, additional table of uniform CF values. This table (in Game 5) provides a vantage point from which all future CF values are in fact uniform, and simplifies reasoning about many of the critical events of interest. Finally, we formally prove honesty and freshness in Game 5 to show the gap between Game 5 and 6 is negligible.

Technical Differences Between [9] and This Paper. In [9] (the classical indifferentiability model), Coron et al. used a simulation strategy similar to ours—simulation via judicious preemptive chain completion—to demonstrate the classical indifferentiability of a constant round Feistel structure. Despite using similar simulation strategy, there are some significant technical differences between our security proof and the proof in [9].

1. **Freshness:** The proof of freshness is challenging in both [9] and our work, but for quite different reasons. The chains in [9] are very short (i.e., have only constant length), and when two of them are intersected, the terms of one chain can easily occupy the "adaptation space" for the other, which hinders freshness. In our case, however, we are not that worried about the intersection of chains since our construction has many more than constant rounds. The difficulty of our freshness proof arises from the subversion algorithm: to prove freshness, we need to rule out the case that when completing a chain, the two adapted terms are queried by some previously evaluated $\widetilde{CF}$.
2. **Honesty:** In the security proof of [9], the authors only need to show freshness and efficiency of the simulation since there is no subversion; they are not required to prove honesty.
3. **Efficiency:** The efficiency proof in [9] is quite straightforward. By contrast, in our case, it is not that obvious how to upper bound the number of the terms generated in the simulation. The difference is again due to the existence of the subversion algorithm. In our case, the chains that are completed are subverted chains, while the classical case has no subversion algorithm and therefore only completes "unsubverted" chains. The evaluation of a subverted chain generates many more terms than the evaluation of an unsubverted chain, which in general, may generate many more chains that trigger completion.

4 Security Proof

In the rest of the paper, we turn the explanation above into a real proof. We first introduce the detailed definition of the simulator.

4.1 The Detailed Definition of the Simulator

The simulator provides an interface $\mathcal{S}.CF(i, x)$ to query the simulated random function CF_i on input x. As mentioned above, for each i the simulator internally maintains a table whose entries are pairs (x, y) of n-bit strings; each such entry intuitively determines a simulated value of CF at a particular point: in particular, if the pair (x, y) appears then any query to $\mathcal{S}.CF(i, x)$ returns the value y. The simulator maintains the natural invariants described previously: responses provided to the distinguisher are always consistent with the table; furthermore, once an entry has been added to the table, it is never removed or changed. Note that in many cases the table will reflect function values that have not been queried by the distinguisher. We denote the ith table by $\mathcal{S}.CF_i$ and write $x \in \mathcal{S}.CF_i$ whenever x is a preimage in this table, often identifying $\mathcal{S}.CF_i$ with the set of preimages stored. When $x \in \mathcal{S}.CF_i$, $CF_i(x)$ denotes the corresponding image. $\mathcal{S}.CF$ is the collection of all these $\mathcal{S}.CF_i$ tables. We use the notation $(i, x) \in \mathcal{S}.CF$ when $x \in \mathcal{S}.CF_i$.

For each i, we additionally define a table $\mathcal{S}.\tilde{CF_i}$ induced implicitly by $\mathcal{S}.CF$. As with $\mathcal{S}.CF_i$, the table $\mathcal{S}.\tilde{CF_i}$ consists of pairs of inputs and outputs of $\tilde{CF_i}$. We write $x \in \mathcal{S}.\tilde{CF_i}$ when all queries generated by evaluation of $\tilde{CF_i}(x)$ are defined in $\mathcal{S}.CF$; naturally, the corresponding function value determines the pair (x, y) in the table. The collection of all of these $\mathcal{S}.\tilde{CF_i}$ is denoted by $\mathcal{S}.\tilde{CF}$. (Note that this table is not maintained explicitly by the simulator, but rather determined implicitly by $\mathcal{S}.CF$.)

Handling Queries to $\mathcal{S}.CF$. On a query $\mathcal{S}.CF(i, x)$, the simulator first checks whether $x \in \mathcal{S}.CF_i$. If so, it answers with $CF_i(x)$. Otherwise the simulator picks a random value y and inserts (x, y) into $\mathcal{S}.CF_i$. (The process above is done by a procedure called $\mathcal{S}.CF^{\text{Inner}}$ which takes input (i, x).) After this, the simulator takes further steps to ensure that its future answers are consistent with the permutation P. Only after this consistency maintenance step is the value y finally returned.

To ensure consistency, the simulator considers all newly generated unsubverted chains with length $\ell/20$ that terminate at the last-queried position; for a newly evaluated term $CF_s(x_s)$, these chains of interest either have the form $(s, x_s, ..., x_{s+\ell/20-1})$ or $(s - \ell/20 + 1, x_{s-\ell/20+1}, ..., x_s)$. Each such detected chain is enqueued by the simulator in a "completion queue," identifying the chain for future completion.

The simulator then repeats the following detection and completion step until the queue is emptied. (When the queue is finally empty, the simulator returns the answer y to the initial query).

1. **Detection Step.** The first chain $c = (s, x_s, ..., x_{s+\ell/20-1})$ is removed from the queue. A procedure called $\mathcal{S}$.HonestyCheck is then run on the chain. The procedure $\mathcal{S}$.HonestyCheck evaluates $\tilde{CF}$ values of the elements of c and generates a four-tuple (s, x_s, x_{s+1}, u) for future completion if all the elements in c are honest. (In fact, not all chains removed from the queue are processed by $\mathcal{S}$.HonestyCheck. A chain removed from the queue is processed by $\mathcal{S}$.HonestyCheck only if it is disjoint with all the chains that are previously processed by $\mathcal{S}$.HonestyCheck and is disjoint with all the previously completed full subverted chains. Any chain that is not processed by $\mathcal{S}$.HonestyCheck is discarded. The procedure that decides whether a chain is going to be discarded or processed by $\mathcal{S}$.HonestyCheck is called $\mathcal{S}$.Check.) In the tuple (s, x_s, x_{s+1}, u), the value s ensures that later the simulator knows that the first value x_s corresponds to CF_s. The value u describes where to adapt (that is, program) the values of CF in order to ensure consistency with the given permutation: this will occur at positions u and $u+1$. The convention for determining u is straightforward: If $s > 3\ell/4$ or $s + \ell/20 - 1 < \ell/4$, then there is "plenty of space around $\ell/2$," and $u = \ell/2$; otherwise, $u = \ell - 10$.
2. **Completion Step.** Finally, the simulator takes the four-tuple (s, x_s, x_{s+1}, u) and *completes* the subverted chain related to (s, x_s, x_{s+1}). Intuitively, this

means that the chain is determined by iteratively determining neighbouring values of $\tilde{CF}(x)$ by evaluating the subversion algorithm and, when necessary, carrying out internal calls to $CF_i()$ in order to answer queries made by that algorithm to the F_i. This iterative process is continued, using P to "wrap around," until the only remaining undetermined values appear at positions u and $u + 1$; at this point, the values at u and $u + 1$ are programmed to ensure consistency. In more detail: Assuming that $u < s$, the completion process (conducted by a procedure called $\mathcal{S}$.Complete) proceeds as follows.

– The initial chain consists of the two adjacent values x_s, x_{s+1}.
– $\tilde{CF}_{s+1}(x_{s+1})$ is determined by simulating the subversion algorithm which generates oracle queries to CF to be answered using $\mathcal{S}.CF$. (Note that this process may enqueue new chains for completion.) The value $x_{s+2} = x_s \oplus \tilde{CF}_{s+1}(x_{s+1})$ is then determined, yielding the enlarged chain (x_s, x_{s+1}, x_{s+2}). This process is repeated until the chain is extended maximally "to the right" so that it has the form $(x_s, x_{s+1}, \ldots, x_\ell, x_{\ell+1})$.
– P^{-1} is then applied to $x_\ell, x_{\ell+1}$ to yield x_0, x_1.
– Starting from the pair (x_0, x_1), this process is repeated, as above, to yield values for $x_2, \ldots, x_u$. Note that $x_u = x_{u-2} \oplus \tilde{CF}(x_{u-1})$ so that $\tilde{CF}(x_u)$ is never evaluated during this process (which is to say that the subversion algorithm is never simulated on x_u).
– Similarly, the original pair x_s, x_{s-1} is extended "to the left" to determine the values $x_{s-1}, ..., x_{u+1}$; as above, x_{u+1} is determined by $x_{u+3} \oplus \tilde{CF}(x_{u+2})$, so that $\tilde{CF}(x_{u+1})$ is never evaluated.
– Then, the simulator defines $CF_u(x_u)$ and $CF_{u+1}(x_{u+1})$ that is consistent with P, i.e., $CF_u(x_u) := x_{u-1} \oplus x_{u+1}$ and $CF_{u+1}(x_{u+1}) := x_u \oplus x_{u+2}$. The game aborts if either of these is defined from a previous action of $\mathcal{S}$. If the game does not abort, the simulator evaluates the subversion algorithm on both x_u and x_{u+1}. During this evaluation, the values $CF_u(x_u)$ and $CF_{u+1}(x_{u+1})$ are already determined; other queries are answered using $\mathcal{S}.CF$ as above. The game aborts if (u, x_u) or $(u+1, x_{u+1})$ is dishonest; otherwise, the chain is a valid subverted chain (and consistent with P).
– A set $\mathcal{S}$.CompletedChains is maintained to store the chains that are completed: for any (i, x_i, x_{i+1}) $(1 \leq i \leq \ell - 1)$, $\mathcal{S}$ updates

$$\mathcal{S}.\text{CompletedChains} := \mathcal{S}.\text{CompletedChains} \cup (i, x_i, x_{i+1}).$$

The alternative case, when $u > s + 1$, is treated analogously.

4.2 Plan of the Proof

To establish crooked indifferentiability, we need to prove that, from the perspective of $\mathcal{D}$, interacting with $(P, \mathcal{S}^P)$ (the ideal world) is indistinguishable from interacting with (C^F, F) (the real world).

Recall that we have three challenges in the security proof:

1. **Freshness and Honesty.** We need to show that the values of $CF_u(x_u)$ and $CF_{u+1}(x_{u+1})$ are always undefined when these values are selected for programing. Moreover, we hope the two terms (u, x_u) and $(u + 1, x_{u+1})$, which are

adapted to ensure consistency are always honest; i.e., $CF_u(x_u) = \tilde{CF}_u(x_u)$ and $CF_{u+1}(x_{u+1}) = \tilde{CF}_{u+1}(x_{u+1})$.

2. **Efficiency.** We need to show that the simulation terminates with high probability when answering a query; i.e., the queue becomes empty after a small (polynomial) number of completions.
3. **Indistinguishability.** Finally, with the two demands above in hand, it is still necessary to show that the simulated world cannot be distinguished from the real world.

Let us define the event that $\mathcal{S}$ aborts as Abort. According to the description of $\mathcal{S}$, Abort happens only when the distinguisher $\mathcal{D}$ finds a chain $(1, x_1, \ldots, x_\ell)$ such that the programmed term, (u, x_u) or $(u+1, x_{u+1})$, has been evaluated before it is programmed or is dishonest. It is easy to see that maintaining freshness and honesty is synonymous with preventing $\mathcal{S}$ from aborting. We will stick to the following plan of the proof to address these challenges.

1. In the first two steps of the proof, we begin by assuming that Challenge 2 has been adequately dealt with, allowing us to focus on resolving Challenges 3 and 1. First, in Sect. 4.3, we aim to establish that resolving Challenge 1 enables us to address Challenge 3. Put differently, we will demonstrate that our construction is crooked indifferentiable if Abort happens negligibly.
2. Second, in Sect. 4.4, we address Challenge 1 under the assumption that Challenge 2 has been successfully addressed. In Theorem 4, we will establish that the likelihood of Abort occurring is negligible given that $\mathcal{S}$ is *efficient*, which means, with overwhelming probability, only a polynomial number of terms are evaluated by $\mathcal{S}$ (or P) when $\mathcal{D}$ interacts with $(P, \mathcal{S}^P)$.
3. Last, in Sect. 4.5, we will address Challenge 2 by showing the efficiency of $\mathcal{S}$ in Theorem 5.

A Simplified Proof. Unfortunately, due to space limitations, we can only provide a "simplified" proof (which is the plan above) in the main body. A complete and more rigorous proof is put in Section B of the full version [21]. In the simplified proof, we omit less critical details while retaining a focus on the primary aspects relevant to the core argument. To assure readers that the essential concepts and outcomes from the complete proof are preserved in the simplified version, we will outline the structure of the complete proof and provide a concise explanation of the distinctions between the two versions.

Compare the Complete and Simplified Proof. In the complete proof, we deal with Challenge 3 by developing a "game transition approach". We introduce four intermediate games to build the connection between the ideal world (the interaction between $\mathcal{D}$ and $(P, \mathcal{S}^P)$) and the real world (the interaction between $\mathcal{D}$ and (C^F, F)). Using the game transition, we clearly analyze the gaps between adjacent games. Summing up these gaps gives the gap between the ideal and real world, which is bounded by the probability of two bad events, $\mathsf{BadComplete}_5$ and $\mathsf{BadEval}_5$. The first and the major bad event $\mathsf{BadComplete}_5$ is same as the bad event Abort we defined above. (In fact, there is a little difference between

BadComplete$_5$ and Abort. BadComplete$_5$ is defined in one of the four intermediate games, while Abort is defined in the interaction between $\mathcal{D}$ and $(P, \mathcal{S}^P)$. Otherwise, the two bad events are same and we can use the same proof to show their probabilities are negligible.) The second and the auxiliary bad event BadEval$_5$ is derived from the game transition, which is used to make the proof rigorous. The missing part in the simplified proof are the four intermediate games in the game transitions and the proof that bounds the probability of BadEval$_5$, which is an auxiliary event.

Although we omit the details of the four intermediate games in the simplified proof, we will provide a concise overview of the central ideas underpinning the game transition approach. This summary will explain why crooked indifferentiability can be reduced to the negligibility of Abort. (See Sect. 4.3) We also want to stress that the efficiency proof of the simulator in the simplified proof (Sect. 4.5) is same as that in the complete proof.

4.3 Relating Crooked Indifferentiability to the Bad Event

To understand how crooked indifferentiability is related to the probability of Abort, we consider a situation where we need to "complete" a chain in the ideal world $(P, \mathcal{S}^P)$ and in the real world (C^F, F).

Suppose in both worlds, we start with an initial table of CF values T_{initial}. Suppose there is an unsubverted chain $c = (s, x_s, ..., x_{s+\ell/20-1})$ in T_{initial} that has passed the test of $\mathcal{S}$.HonestyCheck, which means that all the elements of c are honest. (Without loss of generality, we assume $s + \ell/20 - 1 < \ell/4$. This means, when $\mathcal{S}$ completes c, it adapts the value of CF at $u = \ell/2$.) Now we want to see the gap between the two worlds when generating a subverted full chain c' that contain c.

In the ideal world $(P, \mathcal{S}^P)$, what $\mathcal{S}$ does is the following **Procedure 1**:

1. Generate $\tilde{CF}$ values before the adaption position $\ell/2$ by uniformly selecting CF values as needed: For $i = 2, \ldots, \ell/2$, generate (i, x_i) recursively by defining $x_i := x_{i-2} \oplus \tilde{CF}_{i-1}(x_{i-1})$ for $2 \le i \le \ell/2$ (each CF as needed is evaluated uniformly).
2. Generate $\tilde{CF}$ values after the adaption position $\ell/2 + 1$ by querying P and uniformly selecting CF values as needed: Query P at (x_0, x_1) and receive an (almost) uniform pair of n-bit strings $(x_\ell, x_{\ell+1})$. To generate (i, x_i) $(i = \ell/2, \ldots, \ell)$, recursively define $x_{i-2} := x_i \oplus \tilde{CF}_{i-1}(x_{i-1})$ for $\ell/2 + 3 \le i \le \ell + 1$ (each CF as needed is evaluated uniformly).
3. Adapt CF values at the adaption positions $\ell/2$ and $\ell/2+1$: Define $CF_u(x_{\ell/2}) := x_{\ell/2-1} \oplus x_{\ell/2+1}$ and $CF_{\ell/2+1}(x_{\ell/2+1}) := x_{\ell/2} \oplus x_{\ell/2+2}$. Evaluate $\tilde{CF}_{\ell/2}(x_{\ell/2})$ and $\tilde{CF}_{\ell/2+1}(x_{\ell/2+1})$ (each CF as needed is evaluated uniformly).
4. Abort if freshness or honesty is violated: The game aborts if there is an index j such that $\ell/4 \le j \le 3\ell/4$ and (j, x_j) is in T_{initial} or $\bigcup_{i=1}^{\ell} Q_i(x_i)/Q_j(x_j)$. The game also aborts if there is an index j such that $(\ell/2, x_{\ell/2})$ or $(\ell/2 + 1, x_{\ell/2+1})$ is dishonest.

In the real world (C^F, F), we extend c to a full chain by **Procedure 2**:

1. For $i = 2, \ldots, \ell{+}1$, generate (i, x_i) recursively by defining $x_i := x_{i-2} \oplus \tilde{CF}_{i-1}(x_{i-1})$ for $2 \leq i \leq \ell + 1$ (each CF is evaluated uniformly).
2. Assign $P(x_0, x_1) = (x_\ell, x_{\ell+1})$.

To connect Procedure 1 to Procedure 2, we rewrite Procedure 1 as **Procedure 1'**:

1. For all $x \in \{0,1\}^n$ and $\ell/2 + 2 \leq i \leq \ell$, evaluate $\tilde{CF}_i(x)$ (each CF is evaluated uniformly).
2. Same as step 2 of Procedure 1.
3. Same as step 3 of Procedure 1 except that no additional uniform CF values need to be selected because all needed CF values are already evaluated in Step 1.
4. Same as step 4 of Procedure 1.
5. Same as step 5 of Procedure 1.

Procedure 1 and Procedure 1' are equivalent in the sense that the resulting full chain c' in these two procedure are same.

Slightly changing Procedure 1' gives **Procedure 2'**:

1. Same as step 1 of Procedure 1'.
2. Same as step 2 of Procedure 1'.
3. Select $x_{\ell/2+1}$ and $x_{\ell/2+2}$ uniformly. To generate (i, x_i) (for $i = \ell/2{+}3, \ldots, \ell{+}1$), define $x_i := x_{i-2} \oplus \tilde{CF}_{i-1}(x_{i-1})$ for $\ell/2{+}3 \leq i \leq \ell{+}1$. Assign $P(x_0, x_1) = (x_\ell, x_{\ell+1})$.
4. Same as step 4 of Procedure 1'.

Notice that step 3 of Procedure 2' is equivalent to that of Procedure 1' because the Feistel structure gives a permutation of $2n$-bit strings: selecting a uniform "input" string $(x_{\ell+1}, x_{\ell+2})$ is equivalent to selecting an uniform "output" string $(x_\ell, x_{\ell+1})$. (In fact, they are not perfectly equivalent since in Step 3 of Procedure 1', querying P at (x_0, x_1) does not give a perfectly uniform $(x_\ell, x_{\ell+1})$: P is a random permutation so $(x_\ell, x_{\ell+1})$ is chosen in a way to avoid collision. However, this only causes a negligible difference as we assume T_{initial} contains a polynomial number of terms).

Therefore, the only difference between Procedure 1' and Procedure 2' is that, in Step 5, Procedure 1' aborts when freshness or honesty is violated. And this is indicated by the occurrence of the bad event Abort. Moreover, observe that Procedure 2' is equivalent to Procedure 2 since in both procedures, all CF values are selected uniformly and P values are determined by CF. By combining these observations, it can be inferred that the gap between Procedure 1 and Procedure 2 is bounded by $\Pr[\mathsf{Abort}]$.

4.4 Bounding the Bad Events

In this section, we assume $\mathcal{S}$ is efficient to prove $\Pr[\mathsf{Abort}]$ is negligible. The efficiency of $\mathcal{S}$ will be proved in the next section. In the rest of the paper, all the definitions, lemmas and theorems are in the interaction game between $\mathcal{D}$ and $(P, \mathcal{S}^P)$ unless otherwise specified.

We recall the following example to explain our plan for bounding $\Pr[\mathsf{Abort}]$. During the interaction with the distinguisher, suppose the simulator $\mathcal{S}$ sees an unsubverted chain $c = (s, x_s, ..., x_{s+r})$ that triggers completion. Let us call the current $\mathcal{S}.CF$ table T_{Initial}. For the full chain $c' = (1, x_1, \ldots, x_\ell)$ determined by c ($c \subset c'$), $\mathcal{S}$ hopes that it can find an index u so that (u, x_u) and $(u+1, x_{u+1})$ are undefined before adaptation. It is easy to find an index u so that these two terms are not in T_{Initial}. However, before $\mathcal{S}$ determines x_u and x_{u+1}, it needs to evaluate $\tilde{CF_i}(x_i)$ for $i \neq u, u+1$. And there may exist an index i so that (u, x_u) or $(u+1, x_{u+1})$ is in $Q_i(x_i)$. It is also not obvious how to find u so that (u, x_u) and $(u+1, x_{u+1})$ are honest since the distinguisher can subvert the round functions of any index.

We deal with the challenge in the following three steps.

Step 1: Analysis of unsubverted chains: We introduce the notion of monotone increasing (and decreasing) chains to analyze the property of unsubverted chains. We show that any unsubverted chain is a union of a decreasing chain and an increasing chain. We also show that (Theorem 3), inside a long monotone chain c^*, for any other term (j, x_j) in the "middle area" of c^* and any term $(i, x_i) \in c^*$ $(i \neq j)$, (j, x_j) is honest and $CF_j(x_j)$ is not queried by $\tilde{CF_i}(x_i)$(i.e., $(j, x_j) \notin Q_i(x_i)$).

Step 2: Analysis of subverted chains: We prove all the dishonest terms on a subverted chain are located on an interval shorter than $\ell/12$. As a result, the subverted chain c' can be viewed as an unsubverted chain except for a small dishonest area. That is to say, there always exists a long unsubverted chain $c'' \subset c'$.

Step 3: Bounding the bad event: By combining the two results above, we can deduce the existence of a long monotone chain $c^* \subset c'' \subset c'$. To conclude the proof regarding the negligibility of $\Pr[\mathsf{Abort}]$, we demonstrate that the selection rule for the adaptation terms (u, x_u) and $(u+1, x_{u+1})$ ensures that these two terms fall within the middle area of c^*, which implies honesty and freshness.

- Step 1: Analysis of Unsubverted Chains. To analyze the properties of unsubverted chains, we first introduce the notion of monotone chains.

The Order Function; Monotone Chains. To record the order in which $\mathcal{S}$ sets CF values, we define the following order function O from $\{1, \ldots, \ell\} \times \{0,1\}^n$ to positive integers (with an additional symbol $\perp$):

$$O(i, x) = \begin{cases} t & \text{if } CF_i(x) \text{ is the } t\text{-th evaluated } CF \text{ value by } \mathcal{S}, \\ \perp & \text{if } CF_i(x) \text{ is undefined in } \mathcal{S}.CF. \end{cases}$$

An unsubverted chain $(s, x_s, \ldots, x_{s+r})$ in $\mathcal{S}.CF$ is said to be *monotone increasing* (or *monotone decreasing*) if $O(i, x_i) < O(i+1, x_{i+1})$ for all $s \leq i < s+r$ (or, likewise, $O(j, x_j) > O(j+1, x_{j+1})$ for all $s \leq j < s+r$).

In the rest of the paper, w.l.o.g, we focus our analytic efforts on increasing chains; the results related to increasing chains can be easily transitioned into those related to decreasing chains. In the following lemma, we show that any unsubverted chain is a union of a decreasing chain and an increasing chain.

Lemma 2. *If $\mathcal{S}$ is efficient, then with overwhelming probability, any unsubverted chain $c = (s, x_s, \ldots, x_{s+r})$ in $\mathcal{S}.CF$ will satisfy one of the three conditions below:*

1. *c is increasing,*
2. *c is decreasing,*
3. *There exists an index $s < v < s + r$ such that $(s, x_s, \ldots, x_v)$ is decreasing and $(v, x_v, \ldots, x_{s+r})$ is increasing.*

Proof. It suffices to show that in $\mathcal{S}.CF$ there is no unsubverted length three chain $(s, x_s, x_{s+1}, x_{s+2})$ such that $CF_{s+1}(x_{s+1})$ is evaluated after both $CF_s(s_s)$ and $CF_{s+2}(x_{s+2})$ are evaluated. Suppose that throughout the interaction between $\mathcal{D}$ and $(P, \mathcal{S}^P)$, there are no more than P (= poly(n)) elements in $\mathcal{S}.CF$. Then,

$$\begin{aligned}
&\Pr\left[\begin{array}{l}\text{There is a length 3 chain } (s, x_s, x_{s+1}, x_{s+2}) \\ \text{such that } O(s+1, x_{s+1}) > \max\{O(s+ \\ 2, x_{s+2}), O(s, x_s).\}\end{array}\right] \\
&= \sum_{i=2}^{P} \Pr\left[\begin{array}{l}\text{There is a length 3 chain } (s, x_s, x_{s+1}, x_{s+2}) \\ \text{such that } O(s+1, x_{s+1}) = i > \max\{O(s+ \\ 2, x_{s+2}), O(s, x_s)\}.\end{array}\right] \\
&= \sum_{i=2}^{P} \sum_{\substack{j,k<i \\ j\neq k}} \Pr\left[\begin{array}{l}\text{There is a length 3 chain } (s, x_s, x_{s+1}, x_{s+2}) \text{ such} \\ \text{that } O(s+1, x_{s+1}) = i,\ O(s+2, x_{s+2}) = j \text{ and} \\ O(s, x_s) = k.\end{array}\right] \\
&< \sum_{i=2}^{P} \sum_{\substack{j,k<i \\ j\neq k}} \frac{1}{2^n} < \frac{P^3}{2^n} = \mathsf{negl}(n),
\end{aligned}$$

where the first inequality is based on the fact that and $CF_{s+1}(x_{s+1})$ is selected uniformly and is independent of $CF_s(s_s)$ and $CF_{s+2}(x_{s+2})$. □

Parameters in the Main Theorem. The security parameter ϵ' in Theorem 1 is determined by the last line of the inequality in the proof of Lemma 2, along with Theorem 5, which states that P is bounded by $22q_{\mathcal{D}}(q_{\mathcal{A}} + 1)$.

Next, we will use a sequence of lemmas to establish the following major theorem that describes the nice properties of increasing chains.

Theorem 3. *If $\mathcal{S}$ is efficient, then with overwhelming probability, any unsubverted increasing chain $c = (s, x_s, \ldots, x_{s+r})$ $(r > 8)$ in $\mathcal{S}.CF$ will satisfy:*

1. *for any $0 < i < j$ and $8 < j \leq r$, $(s + j, x_{s+j}) \notin Q_{s+i}(x_{s+i})$;*
2. *for any $7 \leq i < j \leq r$, $(s + i, x_{s+i}) \notin Q_{s+j}(x_{s+j})$.*

Lemma 3. *With overwhelming probability, the following event does not happen: at some point of the game, there exist an unsubverted (or subverted) chain $c = (i, x_i, \ldots, x_j)$ and a length 10 unsubverted chain $c' = (s, y_s, \ldots, y_{s+9})$ in $\mathcal{S}.CF$ such that*

– *for all* $(j, x) \in c$, *$\tilde{CF_i}(x)$ is defined;*
– *c and c' are disjoint;*
– *for each $s \le k \le s+9$, $(k, y_k) \in Q_c$.*

Proof. Consider proving the following stronger statements: Imagine we fill the entire table $\mathcal{S}.CF$ by uniformly selecting all the F values and (a_i, b_i) $(i = 1, \dots, \ell)$. We will prove that with overwhelming probability over the choice of F values and (a_i, b_i), there are not two chains c and c' that satisfy the properties in the lemma.

Let (x_{i+1}, x_{i+2}), (y_s, y_{s+1}) be two pairs of n-bit strings and (i, j, s) be three positive indices. We denote by c the length $(j-i)$ chain starting with $(i+1, x_{i+1}, x_{i+2})$ (without loss of generality, we assume c is a subverted chain for convenience in the rest of the proof) and denote by c' the length 10 unsubverted chain starting with (s, y_s, y_{s+1}). We denote by x_v $(v = i+1, \dots, j)$ the elements of c and denote by y_k $(k = s, \dots, s+9)$ the elements of c'. It is important to note that while x_{i+1}, x_{i+2}, y_s and y_{s+1} are specific n-bit strings, the values of x_v and y_k are currently undetermined. We use x_v and y_k purely to represent the elements of c and c' respectively. The actual values they will take on will be determined by choice of F values and (a_i, b_i). We define the event:

$$E_{i,j,s}(x_{i+1}, x_{i+2}, y_s, y_{s+1}) := \left\{ \begin{matrix} c \text{ and } c' \text{ are disjoint, and for each } s \le k \le s+9,\ (k, y_k) \in \\ Q_c \end{matrix} \right\}.$$

For $s \le t \le s+9$, we also define:

$$E^t_{i,j,s}(x_{i+1}, x_{i+2}, y_s, y_{s+1}) := \left\{ \begin{matrix} c \text{ and } c' \text{ are disjoint, and for each } s \le k \le t,\ (k, y_k) \in \\ Q_c \end{matrix} \right\}.$$

To analyze the probability of $E_{i,j,s}(x_1, x_2, y_s, y_{s+1})$ over the choice of F and (a_i, b_i) $(i = 1, \dots, \ell)$, we consider selecting uniformly the values of $F_i(x)$ for all $i = 1, \dots, \ell$ and $x \in \{0,1\}^n$ and selecting uniformly $a_v \cdot x_v \oplus b_v$ for $v = i, \dots, j$. Since the function $x_v \to a_i \cdot x_v \oplus b_i$ is pairwise independent, the values of $a_k \cdot y_k \oplus b_k (k = s, \dots, s+9)$ are uniformly random. (For convenience, in the following, we will write $E_{i,j,s}$ for $E_{i,j,s}(x_1, x_2, y_s, y_{s+1})$ and $E^t_{i,j,s}$ for $E^t_{i,j,s}(x_1, x_2, y_s, y_{s+1})$.) Over the randomness of $a_k \cdot y_k \oplus b_k$ $(k = s, \dots, s+9)$, we have

$$\begin{aligned}
&\Pr[\,E_{i,j,s}] \\
&= \Pr[E_{i,j,s} \mid E^{s+1}_{i,j,s}\,] \cdot \Pr[E^{s+1}_{i,j,s}] \\
&< \Pr[\,E_{i,j,s} \mid E^{s+1}_{i,j,s}\,] \\
&\quad \cdot (\Pr[\,CF_s(y_s) \in \cup_{v=i}^{j} Q_v(x_v) \mid y_s \ne x_s\,] + \Pr[\,CF_{s+1}(y_{s+1}) \in \cup_{v=i}^{j} Q_v(x_v) \mid y_{s+1} \ne x_{s+1}\,]) \\
&< \Pr[E_{i,j,s} \mid E^{s+3}_{i,j,s}\,] \cdot \Pr[E^{s+3}_{i,j,s} \mid E^{s+1}_{i,j,s}\,] \cdot 2 \cdot (\ell \cdot q_{\mathcal{A}}/2^n) \\
&< \Pr[E_{i,j,s} \mid E^{s+5}_{i,j,s}\,] \cdot \Pr[E^{s+5}_{i,j,s} \mid E^{s+3}_{i,j,s}\,] \cdot (2\ell \cdot q_{\mathcal{A}}/2^n)^2 \\
&< \Pr[E_{i,j,s} \mid E^{s+7}_{i,j,s}\,] \cdot \Pr[E^{s+7}_{i,j,s} \mid E^{s+5}_{i,j,s}\,] \cdot (2\ell \cdot q_{\mathcal{A}}/2^n)^3 \\
&< \Pr[E_{i,j,s} \mid E^{s+7}_{i,j,s}\,] \cdot (2\ell \cdot q_{\mathcal{A}}/2^n)^4 < (2\ell \cdot q_{\mathcal{A}}/2^n)^5\,.
\end{aligned}$$

The lemma is implied by taking the union bound over the choice of (x_1, x_2, y_s, y_{s+1}).

A similar proof can be used to prove the following lemma:

Lemma 4. *With overwhelming probability over the choice of all the F values and (a_i, b_i) $(i = 1, \ldots, \ell)$, there are not a term (i, x_i) and a length 8 unsubverted chain $c = (s, y_s, \ldots, y_{s+7})$ in $\mathcal{S}.CF$ such that $(k, y_k) \in Q_i(x_i)$ for all $k = s, s+2, s+4, s+6$.*

Lemma 5. *If $\mathcal{S}$ is efficient, then with overwhelming probability, for any unsubverted increasing chain $c = (s, x_s, \ldots, x_{s+r})$ in $\mathcal{S}.CF$, if $(s + 2t + 1, x_{s+2t+1}) \in Q_{s+2k}(x_{s+2k})$ (assuming $\tilde{CF}_{s+2k}(x_{s+2k})$ is defined) for some t, k with $0 < 2t+1, 2k \leq r$, then $(s + 2i, x_{s+2i}) \in Q_{s+2k}(x_{s+2k})$ for all $0 < i \leq t$.*

Proof. We give a simple example to show the idea of the proof. Take $s = 1$, $r = 7$, $k = 2$ and $t = 3$ for example. We want to show that for any chain $c = (1, x_1, \ldots, x_8)$, if $(8, x_8) \in Q_5(x_5)$, then with overwhelming probability, $(1 + 2i, x_{1+2i}) \in Q_5(x_5)$ for $i = 1$.

Consider the following two ways of determining a length 8 unsubverted chain:

- **Procedure 1**:
  1. Pick an arbitrary moment in the interaction between $\mathcal{D}$ and $(P, \mathcal{S}^P)$ and abort the game. Denote the table $\mathcal{S}.CF$ at this moment by T_{initial}. Pick a length 2 increasing chain $(1, x_1, x_2)$ in T_{initial} such that it is not a subchain of a length 3 unsubverted chain.
  2. For $2 \leq i \leq 7$, select $CF_i(x_i)$ uniformly, set $x_{i+1} := CF_i(x_i) \oplus x_{i-1}$ and abort the procedure if $(i + 1, x_{i+1})$ is already in the table T_{initial}.
  3. Evaluate $\tilde{CF}_5(x_5)$.
- **Procedure 2**:
  1. Pick an arbitrary moment in the interaction between $\mathcal{D}$ and $(P, \mathcal{S}^P)$ and abort the game. Denote the table $\mathcal{S}.CF$ at this moment by T_{initial}. Pick a length 2 increasing chain $(1, x_1, x_2)$ in T_{initial} such that it is not a subchain of a length 3 unsubverted chain.
  2. Select $CF_2(x_2)$ uniformly and set $x_3 := a_2 \oplus x_1$.
  3. Select 4 uniform n-bit strings a_4, a_5, a_6 and a_7. Set $x_5 := a_4 \oplus x_3$, $x_7 := a_6 \oplus x_5$ and abort the procedure if either of them is in T_{initial}. Set $CF_5(x_5) := a_5$ and $CF_7(x_7) := a_7$.
  4. Evaluate $\tilde{CF}_5(x_5)$.
  5. Select $CF_3(x_3)$ uniformly (use the existing value if it has been evaluated), set $x_4 := CF_3(x_3) \oplus x_2$, $x_6 := a_5 \oplus x_4$, $x_8 := a_7 \oplus x_6$, and abort the procedure if any one of x_4, x_4 and x_8 is in T_{initial}.

A quick thought reveals that the above two procedures are equivalent in terms of the distribution of the chain and, furthermore, the probability they abort is negligible because of Lemma 2. We use the second procedure to analyze the distribution of the first one. In the second procedure, we can see that if $(3, x_3) \notin Q_5(x_5)$, then $CF_3(x_3)$ is still uniform conditioned on $Q_5(x_5)$, which implies that $x_8 = a_7 \oplus x_6 = a_7 \oplus a_5 \oplus x_4 = a_7 \oplus a_5 \oplus CF_3(x_3) \oplus x_2$ is uniform. Therefore, if $(3, x_3) \notin Q_5(x_5)$, $(8, x_8) \in Q_5(x_5)$ with negligible probability.

The full proof can be achieved by replacing the concrete numbers in the last example by more general parameters s, r, k and t and taking the union bound over the various values of these parameters.

Lemma 6. *If $\mathcal{S}$ is efficient, then with overwhelming probability, for any unsubverted increasing chain $c = (s, x_s, \ldots, x_{s+r})$ in $\mathcal{S}.CF$ and any index i, j with $0 < i < j$ and $8 < j \leq r$, $(s + j, x_{s+j}) \notin Q_{s+i}(x_{s+i})$ (if $\tilde{CF}_{s+i}(x_{s+i})$ is defined).*

Proof. Without loss of generality, assume $i = 0$. Suppose $(s + j, x_{s+j}) \in Q_s(x_s)$. Notice that $(s+j-1, x_{s+j-1}) \in Q_{s+i}(x_{s+i})$ with overwhelming probability because otherwise the randomness of $CF_{s+j-1}(x_{s+j-1})$ will cause the event $(s + j, x_{s+j}) \notin Q_s(x_s)$. Then,

– if j is odd, since $j > 8$ and $(s + j, x_{s+j}) \in Q_s(x_s)$, by Lemma 5, $(s + 2k, x_{s+2k}) \in Q_s(x_s)$ for $k = 1, 2, 3, 4$. This contradicts Lemma 4.
– if j is even, since $j > 8$ and $(s + j - 1, x_{s+j-1}) \in Q_s(x_s)$, by Lemma 5, $(s + 2j, x_{s+2j}) \in Q_s(x_s)$ for $j = 1, 2, 3, 4$, which contradicts with Lemma 4. □

Lemma 7. *If $\mathcal{S}$ is efficient, then with overwhelming probability, for any unsubverted increasing chain $c = (s, x_s, \ldots, x_{s+r})$ in $\mathcal{S}.CF$, if $(s + 2t, x_{s+2t}) \in Q_{s+k}(x_{s+k})$ (assuming $\tilde{CF}_{s+k}(x_{s+k})$ is defined) for some t, k with $0 < 2t < k \leq r$, then $(s + 2i - 1, x_{s+2i-1}) \in Q_{s+k}(x_{s+k})$ for all $0 < i \leq t$.*

Proof. The proof of the lemma is similar to that of Lemma 5. Consider the example where $s = 1$, $r = 8$, $t = 2$ and $k = 8$. We want to show that for any chain $c = (1, x_1, \ldots, x_9)$, if $(5, x_5) \in Q_9(x_9)$, then with overwhelming probability, $(2i, x_{2i}) \in Q_9(x_9)$ for $i = 1$.

Consider the following two ways of determining a length 9 unsubverted chain:

– **Procedure 1**:
  1. Pick an arbitrary moment in G_5 and abort the game. Denote the table $\mathcal{M}^3.CF$ at this moment by T_{initial}. Pick a length 2 increasing chain $(1, x_1, x_2)$ in T_{initial} such that it is not a subchain of a length 3 unsubverted chain.
  2. For $2 \leq i \leq 8$, select $CF_i(x_i)$ uniformly, set $x_{i+1} := CF_i(x_i) \oplus x_{i-1}$ and abort the procedure if $(i + 1, x_{i+1})$ is already in the table T_{initial}.
  3. Evaluate $\tilde{CF}_9(x_9)$.
– **Procedure 2**:
  1. Pick an arbitrary moment in G_5 and abort the game. Denote the table $\mathcal{M}^3.CF$ at this moment by T_{initial}. Pick a length 2 increasing chain $(1, x_1, x_2)$ in T_{initial} such that it is not a subchain of a length 3 unsubverted chain.
  2. Select 3 uniform n-bit strings a_3, a_4 and a_5. Set $x_4 := a_3 \oplus x_2$, $x_6 := a_5 \oplus x_4$ and aborts the procedure if either of them is in T_{initial}. Set $CF_4(x_4) := a_4$ and $CF_6(x_6) := a_6$.
  3. Select x_7, x_8 and x_9 uniformly and aborts the procedure if any one of them is in T_{initial}. Set $CF_7(x_7) := x_6 \oplus x_8$ and $CF_8(x_8) := x_7 \oplus x_9$.
  4. Evaluate $\tilde{CF}_9(x_9)$.
  5. Select $CF_2(x_2)$ uniformly(use the existing value if it has been evaluated), set $x_3 := CF_2(x_2) \oplus x_1$, $x_5 := a_4 \oplus x_3$, $CF_6(x_6) := x_7 \oplus x_5$, and aborts the procedure if either x_3 or x_5 is in T_{initial}.

A quick thought reveals that the above two procedures are equivalent in terms of the distribution of the chain (and, furthermore, the probability they abort is negligible because of Lemma 2). We use the second procedure to analyze the distribution of the first one. In the second procedure, we can see that if $(2, x_2) \notin Q_9(x_9)$, then $CF_2(x_2)$ is still uniform conditioned on $Q_9(x_9)$, which implies that $x_5 = a_4 \oplus x_3 = a_4 \oplus a_2 \oplus x_1$ is uniform. Therefore, if $(2, x_2) \notin Q_9(x_9)$, $(5, x_5) \in Q_9(x_9)$ with negligible probability.

The formal proof can be achieved by replacing the concrete numbers in the last example by more general parameters s, r, t and k and taking the union bound over the various values of these parameters. □

Following directly from Lemma 4 and Lemma 7, we get:

Lemma 8. *If $\mathcal{S}$ is efficient, then with overwhelming probability, for any unsubverted increasing chain $c = (s, x_s, \ldots, x_{s+r})$ in $\mathcal{S}.CF$ and any index i, j with $7 < i < j \leq r$, $(s + i, x_{s+i}) \notin Q_{s+j}(x_{s+j})$(if $\tilde{CF}_{s+j}(x_{s+j})$ is defined).*

Theorem 3 follows from the combination of Lemma 6 and Lemma 8.

- Step 2: Analysis of Subverted Chains. Now we turn our attention to subverted chains. We want to show that although, in general, there are some dishonest terms on a subverted chain, all of them gather in a small area.

Lemma 9. *If $\mathcal{S}$ is efficient, then with overwhelming probability, there does not exist an unsubverted increasing chain $c = (i, x_i, \ldots, x_{i+8})$ in $\mathcal{S}.CF$ such that $\tilde{CF}_{i+8}(x_{i+8})$ is defined in $\mathcal{S}.CF$ and $(i + 8, x_{i+8})$ is dishonest.*

Proof. We say the distinguisher $\mathcal{D}$ wins the interaction game with $(P, \mathcal{S}^P)$ if it is able to find an unsubverted increasing chain c in $\mathcal{S}.CF$ that satisfies the property in the lemma. By Lemma 6, the probability that there is a length-9 unsubverted increasing chain $c = (i, x_i, \ldots, x_{i+8})$ with $(i + 7, x_{i+7}) \in Q_{i+8}(x_{i+8})$ $(\tilde{CF}_{i+8}(x_{i+8}))$ is negligible. We denote this negligible probability by δ. To show the probability that $\mathcal{D}$ wins is negligible, consider the following experiment with a distinguisher $\mathcal{D}^*$:

EXP*

1. $\mathcal{D}^*$ takes an arbitrary moment of $\mathcal{S}$, stops the game and selects an arbitrary length-2 increasing chain (i, x_i, x_{i+1}) in $\mathcal{S}.CF$ such that $CF_{i+2}(x_{i+2})$ is not evaluated for $x_{i+1} := x_i \oplus CF_i(x_i)$.
2. Then, $\mathcal{D}^*$ extends (i, x_i, x_{i+1}) to $(i, x_i, \ldots, x_{i+7})$ by iteratively evaluating $CF_{j-1}(x_{j-1})$ (selected uniformly) and $x_j := x_{j-2} \oplus CF_{j-1}(x_{j-1})$ for $i + 3 \leq j \leq i + 7$. The experiment aborts if $CF_{i+7}(x_{i+7})$ is already evaluated.
3. For any term (j, y), if $CF_j(y)$ is still unevaluated and $(j, y) \neq (i + 7, x_{i+7})$, $\mathcal{D}^*$ selects $CF_j(y)$ uniformly.
4. Finally $\mathcal{D}^*$ selects $CF_{i+7}(x_{i+7})$ and check if $(i + 8, x_{i+8})$ is dishonest for $x_{i+8} := x_{i+6} \oplus CF_{i+7}(x_{i+7})$.
5. $\mathcal{D}^*$ wins **Exp*** if the experiment does not abort in Step 2 and $(i+8, x_{i+8})$ is dishonest.

To prove $\mathcal{D}$ wins negligibly, it is sufficient to show the probability that the experiment aborts in Step 2 or $\mathcal{D}^*$ wins is negligible. We also stress that although **Exp*** is not the interaction between $\mathcal{D}$ and $(P, \mathcal{S}^P)$, the lemmas we proved in this section can still be applied because all the *CF* values here are also selected uniformly and independently.

$$\begin{aligned}
&\Pr_{\mathbf{Exp^*}}[\text{The experiment aborts in Step 2 or } \mathcal{D}^* \text{ wins.}] \\
&\leq \Pr_{\mathbf{Exp^*}}[\text{The experiment aborts in Step 2.}] + \Pr_{\mathbf{Exp^*}}\left[\begin{array}{l}\mathcal{D}^* \text{ wins and there are at least } \sqrt{\delta}2^n\ n\text{-bit} \\ \text{strings } x \text{ such that } (i+7, x_{i+7}) \in Q_{i+8}(x).\end{array}\right] \\
&\qquad + \Pr_{\mathbf{Exp^*}}\left[\begin{array}{l}\mathcal{D}^* \text{ wins and there are fewer than } \sqrt{\delta}2^n\ n\text{-bit strings } x \\ \text{such that } (i+7, x_{i+7}) \in Q_{i+8}(x).\end{array}\right] \\
&< \mathsf{negl}(n) + \Pr_{\mathbf{Exp^*}}\left[\mathcal{D}^* \text{ wins and there are at least } \sqrt{\delta}2^n\ n\text{-bit strings } x \text{ such that } (i+7, x_{i+7}) \in Q_{i+8}(x).\right] \\
&\qquad + \Pr_{\mathbf{Exp^*}}\left[\mathcal{D}^* \text{ wins.} \;\middle|\; \begin{array}{l}\text{There are fewer than } \sqrt{\delta}2^n\ n\text{-bit strings } x \text{ such that } (i+7, x_{i+7}) \in \\ Q_{i+8}(x).\end{array}\right] \\
&< \mathsf{negl}(n) + \sqrt{\delta} + \Pr_{\mathbf{Exp^*}}\left[\begin{array}{l}(i+8, x_{i+8}) \text{ is dishonest and} \\ (i+7, x_{i+7}) \in Q_{i+8}(x_{i+8}).\end{array} \;\middle|\; \begin{array}{l}\text{There are fewer than } \sqrt{\delta}2^n\ n\text{-bit strings} \\ x \text{ such that } (i+7, x_{i+7}) \in Q_{i+8}(x).\end{array}\right] \\
&\qquad + \Pr_{\mathbf{Exp^*}}\left[\begin{array}{l}(i+8, x_{i+8}) \text{ is dishonest and} \\ (i+7, x_{i+7}) \notin Q_{i+8}(x_{i+8}).\end{array} \;\middle|\; \begin{array}{l}\text{There are fewer than } \sqrt{\delta}2^n\ n\text{-bit} \\ \text{strings } x \text{ such that } (i+7, x_{i+7}) \in \\ Q_{i+8}(x).\end{array}\right] \\
&< \mathsf{negl}(n) + \sqrt{\delta} + \sqrt{\delta} + \epsilon = \mathsf{negl}(n). \square
\end{aligned}$$

Definition 2 (Bad region). *For a subverted chain $c = (s, x_s, \ldots, x_{s+r})$ in $\mathcal{S}.CF$, we say a subchain $(i, x_i, \ldots, x_j)$ $(s \leq i < j \leq s+r)$ of c is a* bad region *of c if there is no sequence of 14 consecutive elements $(k, x_k, \ldots, x_{k+13})$ $(i \leq k \leq j-13)$ that are honest.*

For a subverted chain $c = (s, x_s, \ldots, x_{s+r})$ in $\mathcal{S}.CF$, we say two bad regions of c, $(i, x_i, \ldots, x_j)$ and $(i', x_{i'}, \ldots, x_{j'})$ $(i < i', j < j')$ are separated if the subchain $(i, x_i, \ldots, x'_j)$ of c is not a bad region of c.

Lemma 10. *If $\ell > 337n/\log(1/\epsilon)$, then in the interaction game between $\mathcal{D}$ and $(P, \mathcal{S}^P)$, with overwhelming probability, there does not exist a subverted chain $(s, x_s, \ldots, x_{s+r})$ in $\mathcal{S}.CF$ such that it has a bad region with length greater than $\ell/12$.*

Proof. Consider proving the following stronger statement: with overwhelming probability over the uniform choice of (a_i, b_i) $(i = 1, \ldots, \ell)$ and values of $F_i(x)$ for all $i = 1, \ldots, \ell$ and $x \in \{0,1\}^n$, there is no bad region with length greater than $\ell/12$. Imagine we select $F_i(x)$ for all $i = \{1, \ldots, \ell\}$ and $x \in \{0,1\}^n$ and leave a_i and b_i undetermined. Then, over the randomness of the choice of a_i and b_i, we have

$$\begin{aligned}
&\Pr[\text{There is a subverted chain } c \text{ with a bad region longer than } \ell/12.]\\
&= \sum_{i=1}^{\ell} \Pr\left[\begin{array}{l}\text{There is a subverted chain } c \text{ with a bad region longer than } \ell/12 \text{ and the bad region begins at}\\ \text{index } i.\end{array}\right]\\
&= \sum_{i=1}^{\ell} \sum_{x,x' \in \{0,1\}^n} \Pr\left[\begin{array}{l}\text{There is a subverted chain } c \text{ with a bad region longer than } \ell/12. \text{ The bad region begins}\\ \text{at index } i \text{ and its first two elements are } (i,x) \text{ and } (i+1,x').\end{array}\right]\\
&< \sum_{i=1}^{\ell} \sum_{x,x' \in \{0,1\}^n} \Pr\left[\begin{array}{l}\text{There is a subverted chain } c=(i,x,x',\ldots,x_r) \text{ such that its first element has index } i \text{ and } r-\\ i > \ell/12. \text{ Moreover, for any length 14 subchain of } c \text{ in the form of } (14k, x_{14k},\ldots,x_{14k+13}),\\ \text{at least one of 14 elements is dishonest.}\end{array}\right]\\
&< \sum_{i=1}^{\ell} (2^n)^2 \cdot (14\epsilon)^{\ell/168-1} = \ell \cdot 2^{2n} \cdot (14\epsilon)^{\ell/168-1} = \mathsf{negl}(n).
\end{aligned}$$

Lemma 11. *If $\mathcal{S}$ is efficient, then with overwhelming probability, there is no subverted chain $c = (s, x_s, \ldots, x_{s+r})$ in $\mathcal{S}.CF$ that has two separated bad regions.*

Proof. The lemma is implied directly by Lemma 2 and Lemma 9.

- Step 3: Bounding the Bad Event. Now we put together the properties of unsubverted and subverted chains above to show the main theorem:

Theorem 4. *If $\mathcal{S}$ is efficient, the probability that* Abort *happens is negligible.*

Proof. Due to space limitations, we only give a high-level description of how integrating the properties of chains above gives the negligibility of Abort.

Imagine we start with an initial table of CF values T_{initial}. Suppose there is an unsubverted chain $c = (s, x_s, ..., x_{s+\ell/20-1})$ in T_{initial} that has passed the test of $\mathcal{S}$.HonestyCheck. This indicates that all the elements of c are honest. Without loss of generality, here we assume $s + \ell/20 - 1 < \ell/4$. Consequently, when $\mathcal{S}$ completes c, it adapts the value of CF at $u = \ell/2$.

Now we prove freshness and honesty when completing the chain c. We denote by $c' = (1, x_1, ..., x_\ell)$ the full chain that contains c.

- Case 1: There exists a dishonest term (j, x_j) in $(s, x_s, ..., x_\ell)$. In this case, by Lemma 9 and 10, $j < (s + \ell/20 - 1) + 8 + \ell/12 < \ell/4 + 8 + \ell/12 = \ell/3 + 8$. This means the chain $c'' = (\ell/3 + 8, x_{\ell/3+8}, ..., x_\ell)$ is honest. Again by Lemma 9, the chain $c^* = (\ell/2 - 10, x_{\ell/2-10}, ..., x_\ell)$ is increasing. Therefore, the CF values in the adaption positions are honest. They also satisfy freshness since: 1. they are not in Q_{c^*} because of Theorem 3; 2. they are uniform and are therefore outside of T_{initial} and Q_{c'/c^*}.
- Case 2: There does not exist a dishonest term (j, x_j) in $(s, x_s, ..., x_\ell)$. The proof in this case is simply a subset of that of Case 1.

4.5 Efficiency of $\mathcal{S}$

In this section, we are going to show that the number of the elements in $\mathcal{S}.CF$ is bounded by a polynomial function if the distinguisher $\mathcal{D}$ makes at most $q_{\mathcal{D}}$ ($q_{\mathcal{D}}$ is polynomial) queries to CF or the ideal object.

Lemma 12. *If $\mathcal{S}$ is efficient, then with overwhelming probability, there is not a chain $c = (1, w_1, \ldots, w_\ell)$ and three pairwise disjoint increasing chains $c_1 = (i, x_i, \ldots, x_{i+10})$, $c_2 = (j, y_j, \ldots, y_{j+10})$ and $c_3 = (k, z_k, \ldots, z_{k+10})$ in $\mathcal{S}.CF$, such that*

- *for all $(i, x) \in c$, $\tilde{CF_i}(x)$ is defined;*
- *c is disjoint with c_1, c_2, c_3;*
- *$(i + 10, x_{i+10}), (j + 10, y_{j+10}), (k + 10, z_{k+10}) \in Q_c$.*

Proof. According to Lemma 3, if $(i + 10, x_{i+10}) \in Q_c$, then there exists an index m $(i + 1 \leq m \leq i + 9)$ such that in the length 3 monotone increasing chain (x_{i-1}, x_i, x_{i+1}), $(i, x_i) \notin Q_c$ but $(i+1, x_{i+1}) \in Q_c$. Now we turn this observation into a proof.

Consider the following experiment in $\mathcal{S}$. Take an arbitrary pair of n-bit strings (w_1, w_2). $\mathcal{D}$ tries to find a subverted chain c starting with (w_1, w_2) (w.l.o.g., we only consider subverted chain for convenience) and a length 3 increasing chain (x_{i-1}, x_i, x_{i+1}) s.t., $(i, x_i) \notin Q_c$ and $(i + 1, x_{i+1}) \in Q_c$. We show the probability that $\mathcal{D}$ wins is negligible $(\ell q_{\mathcal{A}}/s^n)$: Suppose, without loss of generality, $\mathcal{D}$ queries all the elements in Q_c at the beginning of $\mathcal{S}$. At some moment of the experiment, $\mathcal{D}$ will select a pair of terms $(i-1, x_{i-1}, x_i)$ as the starting pair of target length 3 chain. It is easy to see that, since $(i, x_i) \notin Q_c$, $(i + 1, x_{i+1}) \in Q_c$ with probability not greater than $\text{poly}(n) \cdot \ell q_{\mathcal{A}}/2^n$, where $\text{poly}(n)$ denotes the upper bound of the number of the terms in $\mathcal{S}.CF$.

For any pair (w_1, w_2), we define the event:

$$E(w_1, w_2) := \left\{ \begin{array}{l} \text{There are three monotone increasing unsub-} \\ \text{verted chains } c_1 = (i, x_i, \ldots, x_{i+10}),\ c_2 = \\ (j, y_j, \ldots, y_{j+10}) \text{ and } c_3 = (k, z_k, \ldots, z_{k+10}) \text{ in} \\ \mathcal{S}.CF, \text{ such that } c \text{ is disjoint with } c_1, c_2, c_3 \text{ and} \\ (i + 10, x_{i+10}), (j + 10, y_{j+10}), (k + 10, z_{k+10}) \in Q_c, \\ \text{where } c \text{ is the subverted starting with } (w_1, w_2) \end{array} \right\}$$

Finally we have

$$\sum_{(w_1, w_2) \in \{0,1\}^{2n}} \Pr[E(w_1, w_2)] < 2^{2n} \cdot (\text{poly}(n) \cdot \ell q_{\mathcal{A}}/2^n)^3 = \mathsf{negl}(n). \square$$

Lemma 13. *Suppose $\mathcal{S}$ is efficient. Let C_{11} be a set of length 11 increasing chains and c be a chain in $\mathcal{S}.CF$ such that c is disjoint with any element in C_{11}, and for all $(i, x) \in c$, $\tilde{CF_i}(x)$ is defined. Then, with overwhelming probability, there are at most 20 chains $c' = (i, x_i, \ldots, x_{i+10}) \in C_{11}$ such that $(i + 10, x_{i+10}) \in Q_c$.*

Proof. Suppose there are 21 chains in C_{11} that satisfy the property in the lemma. Notice that for each length 11 chain c', there are at most 9 other length 11 chains that are not disjoint with c'. Then, among the 21 chains satisfying the property in the lemma, we can find 3 pairwise disjoint chains. This contradicts Lemma 12.

Definition 3 (Order of a chain). *We define the order of an unsubverted chain $c = (s, x_s, \ldots, x_{s+r})$ in $\mathcal{S}.CF$ to be:*

$$O(c) := \min_{k=s,\ldots,s+r-1} \Big\{ \max\{O(k, x_k), O(k+1, x_{k+1})\} \Big\}$$

Intuitively speaking, the order of a chain describes the time when a chain is "determined."

Lemma 14. *Suppose $\mathcal{S}$ is efficient. Let C_{Disj} be a set of pairwise disjoint unsubverted chains with length greater than or equal to 4 in $\mathcal{S}.CF$. Define the set A to be the set of the elements of the chains in C_{Disj}. Then, with overwhelming probability, $|A| \geq \sum_{c \in C_{Disj}} (L(c) - 3)$.*

Proof. For any $c \in C_{\text{Disj}}$ and a term (i, x) in c, we say (i, x) is *original* in c if there does not exist a different element $c' \in C_{\text{Disj}}$ such that c and c' intersects at (i, x) and $O(c) \geq O(c')$. Notice that a term (i, x) can be original in at most one chain.

Now we are going to show that, with overwhelming probability, each element in C_{Disj} contains at most 3 non-original terms. Suppose there is a chain $c = (s, x_s, \ldots, x_{s+r})$ that has four non-original terms. Then there are two non-original elements, (i, x_i) and (j, x_j), such that $s \leq i < j - 2 \leq s + r - 2$. Because of Lemma 2, w.l.o.g, we assume

$$O(i, x_i) > O(i+1, x_{i+1}) > O(i+2, x_{i+2}).$$

Since (i, x_i) is non-original in c, there is a chain $c' \neq c$ such that $(i, x_i) \in c'$ and $O(c) \geq O(c')$. Since c and c' are disjoint, $(i+1, x_{i+1}) \notin c'$. Then, since $O(c) \geq O(c')$ and $O(i+1, x_{i+1}) > O(i+2, x_{i+2})$, we have $O(c) > O(c')$, which means $(i+1, x_{i+1})$ is not evaluated when c' has been determined. Finally, because $\mathcal{S}.CF(i+1, x_{i+1})$ is selected uniformly, $(i, x_i) \in c'$ with negligible probability. A contradiction.

Going back to the proof of the lemma, since each term is original in at most one chain and each chain in C_{Disj} has all but 3 original elements, $|A|$ is lower bounded by the sum of the original terms in the elements of C_{Disj}, which is not less than $\sum_{c \in C_{\text{Disj}}} (L(c) - 3)$.

Theorem 5. *For any positive integer $k \leq q_{\mathcal{D}}$, with overwhelming probability, at the end of the k-th round of $\mathcal{S}$, there are fewer than $(22q_{\mathcal{A}} + 1)k$ terms in $\mathcal{S}.CF$.*

Remark. In the proof of Theorem 5, we will make use of Lemma 13 and Lemma 14. However, these lemmas already take efficiency of $\mathcal{S}$ as their assumptions. To reassure the reader that there is not a circular argument here, we imagine that the k-th round of the game is forced to end when $\mathcal{S}.CF$ contains more than $(22q_{\mathcal{A}} + 1)k$ elements. In this way, we can also feel free to reason about $\mathcal{S}.CF[k]$.

Proof. In $\mathcal{S}.CF[k]$, for any unsubverted chain c, we call c a *generator* if c was processed by the procedure $\mathcal{S}$.HonestCheck. We denote by C_G the set of generators. We define a function g from $C_{\text{FComp}}[k]$ to C_G: for each $c_1 \in$

$C_{\mathrm{FComp}}[k]$ and $c \in C_G$, we say $g(c_1) = c$ if $c_1 \subset c$. Define $G := \{(i, x) \mid$ there is $c \in C_G$ such that $(i, x) \in c.\}$.

Since C_G is a set of pairwise disjoint chains, by Lemma 14,

$$|G| \geq \sum_{c \in C_G} (L(c) - 3) = (\ell/20 - 3) \cdot |C_G|. \tag{2}$$

To understand the structure of G, we define several subsets of G. We say a point $(i, x) \in G$ is a *tail point* if there is an increasing $c_2 = (s, x_s, \ldots, x_{s+10})$ in $\mathcal{S}.CF$ (w.l.o.g., we only consider the increasing case) and a chain $c \in C_G$ such that $(i, x) = (s + 10, x_{s+10})$ and $c_2 \subset c$. We say a point $(i, x) \in G$ is a *head point* if it is not a tail point. We denote the sets of the head points and tail points by G_{Head} and G_{Tail}, respectively. For any point $(i, x) \in G_{\mathrm{Tail}}$ and any chain $c \in C_G$, we say c covers (i, x) ($(i, x) \notin c$) if $(i, x) \in Q_c$ or $(i, x) \in Q_{g^{-1}(c)}$ (if c has a preimage in function g). We define G_{Query} to be the set of the points in G_{Tail} that are not covered by any element in C_G. Notice that any element in G_{Query} was queried directly by the distinguisher $\mathcal{D}$. Our goal is to show $|G_{\mathrm{Query}}|$ is big.

By Lemma 2, the number of the elements in G_{Head} is easily bounded by

$$|G_{\mathrm{Head}}| \leq 19 \cdot |C_G|. \tag{3}$$

By Lemma 13

$$|G_{\mathrm{Tail}}/G_{\mathrm{Query}}| \leq 20 \cdot |C_G|. \tag{4}$$

Summarizing Eq. 2, 3 and 4, we have

$$|G_{\mathrm{Query}}| = |G| - |G_{\mathrm{Head}}| - |G_{\mathrm{Tail}}/G_{\mathrm{Query}}| \geq (\ell/20 - 3)|C_G| - 19|C_G| - 20|C_{Fu}| = (\ell/20 - 42)|C_G|.$$

This implies that

$$\begin{aligned} |\mathcal{S}.CF[k]| &\leq \ell \cdot q_{\mathcal{A}} \cdot |C_G| + k \leq \ell \cdot q_{\mathcal{A}} \cdot |G_{\mathrm{Query}}|/(\ell/20 - 42) + k \\ &\leq \ell \cdot q_{\mathcal{A}} \cdot k/(\ell/20 - 42) + k \leq \ell \cdot q_{\mathcal{A}} \cdot k/(\ell/22) + k = (22q_{\mathcal{A}} + 1)k. \end{aligned}$$

We remark that all the statements in the proof are true with overwhelming probability.

5 Conclusions and Open Problems

In this work, we answer an open problem in [19,20] and analyze the classical Feistel structure under the crooked-indifferentiability framework that can give a better construction for correcting subverted random function/permutation to a good random permutation.

There are still many interesting questions remain to be explored: broader applications of crooked-indifferentiability, and whether we can have a truly practical construction.

Acknowledgements. We thank anonymous reviewers for valuable comments. Jiadong Zhu was supported in part by the National Natural Science Foundation of China Grants No. 62325210, 62272441.

References

1. Bellare, M., Paterson, K.G., Rogaway, P.: Security of symmetric encryption against mass surveillance. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. Part I, volume 8616 of LNCS, pp. 1–19. Springer, Heidelberg (2014)
2. Bhattacharyya, R., Nandi, M., Raychaudhuri, A.: Crooked indifferentiability of enveloped xor revisited. In INDOCRYPT **2021**, 73–92 (2021)
3. M. Blum. Designing programs that check their work, November 1988. Technical Report TR-88-009, International Computer Science Institute. Available at http://www.icsi.berkeley.edu/pubs/techreports/tr-88-009.pdf
4. M. Blum and S. Kannan. Designing programs that check their work. In *21st ACM STOC*, pages 86–97. ACM Press, May 1989
5. M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *22nd ACM STOC*, pages 73–83. ACM Press, May 1990
6. Chow, S.S.M., Russell, A., Tang, Q., Yung, M., Zhao, Y., Zhou, H.-S.: Let a non-barking watchdog bite: Cliptographic signatures with an offline watchdog. In: Lin, D., Sako, K. (eds.) PKC 2019. Part I, volume 11442 of LNCS, pp. 221–251. Springer, Heidelberg (2019)
7. Coretti, S., Dodis, Y., Guo, S., Steinberger, J.P.: Random oracles and non-uniformity. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part I, volume 10820 of LNCS, pp. 227–258. Springer, Heidelberg, Apr. / (2018)
8. Coron, J.-S., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-Damgård revisited: How to construct a hash function. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 430–448. Springer, Heidelberg (2005)
9. Coron, J.-S., Holenstein, T., Künzler, R., Patarin, J., Seurin, Y., Tessaro, S.: How to build an ideal cipher: The indifferentiability of the Feistel construction. Journal of Cryptology **29**(1), 61–114 (2016)
10. Dachman-Soled, D., Katz, J., Thiruvengadam, A.: 10-round Feistel is indifferentiable from an ideal cipher. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. Part II, volume 9666 of LNCS, pp. 649–678. Springer, Heidelberg (2016)
11. Dai, Y., Steinberger, J.P.: Indifferentiability of 8-round Feistel networks. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. Part I, volume 9814 of LNCS, pp. 95–120. Springer, Heidelberg (2016)
12. Dodis, Y., Ganesh, C., Golovnev, A., Juels, A., Ristenpart, T.: A formal treatment of backdoored pseudorandom generators. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. Part I, volume 9056 of LNCS, pp. 101–126. Springer, Heidelberg (2015)
13. Dodis, Y., Guo, S., Katz, J.: Fixing cracks in the concrete: Random oracles with auxiliary input, revisited. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part II, volume 10211 of LNCS, pp. 473–495. Springer, Heidelberg, Apr. / (2017)
14. Dodis, Y., Puniya, P.: On the relation between the ideal cipher and the random oracle models. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 184–206. Springer, Heidelberg (2006)
15. Dodis, Y., Puniya, P.: Feistel networks made public, and applications. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 534–554. Springer, Heidelberg (2007)
16. Maurer, U.M., Renner, R., Holenstein, C.: Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 21–39. Springer, Heidelberg (2004)

17. Russell, A., Tang, Q., Yung, M., Zhou, H.-S.: Cliptography: Clipping the power of kleptographic attacks. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. Part II, volume 10032 of LNCS, pp. 34–64. Springer, Heidelberg (2016)
18. Russell, A., Tang, Q., Yung, M., Zhou, H.-S.: Generic semantic security against a kleptographic adversary. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017, pp. 907–922. ACM Press, Oct. / (2017)
19. Russell, A., Tang, Q., Yung, M., Zhou, H.-S.: Correcting subverted random oracles. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. Part II, volume 10992 of LNCS, pp. 241–271. Springer, Heidelberg (2018)
20. A. Russell, Q. Tang, M. Yung, H.-S. Zhou, and J. Zhu. Correcting subverted random oracles, 2021. https://eprint.iacr.org/2021/042
21. A. Russell, Q. Tang, and J. Zhu. Crooked indifferentiability of the feistel construction. Cryptology ePrint Archive, Paper 2024/1456, 2024
22. Tang, Q., Yung, M.: Cliptography: Post-snowden cryptography. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017, pp. 2615–2616. ACM Press, Oct. / (2017)
23. Young, A., Yung, M.: The dark side of "black-box" cryptography, or: Should we trust capstone? In: Koblitz, N. (ed.) CRYPTO'96. LNCS, vol. 1109, pp. 89–103. Springer, Heidelberg (1996)
24. Young, A., Yung, M.: Kleptography: Using cryptography against cryptography. In: Fumy, W. (ed.) EUROCRYPT'97. LNCS, vol. 1233, pp. 62–74. Springer, Heidelberg (1997)

Provable Security of Linux-DRBG in the Seedless Robustness Model

Woohyuk Chung[1](✉), Hwigyeom Kim[2], Jooyoung Lee[1], and Yeongmin Lee[3]

[1] KAIST, Daejeon, Korea
hephaistus@kaist.ac.kr, hicalf@kaist.ac.kr
[2] Norma Inc., Seoul, Korea
[3] DESILO Inc., Seoul, Korea
yeongmin.lee@desilo.ai

Abstract. This paper studies the provable security of the deterministic random bit generator (DRBG) utilized in Linux 6.4.8, marking the first analysis of Linux-DRBG from a provable security perspective since its substantial structural changes in Linux 4 and Linux 5.17. Specifically, we prove its security up to $O(\min\{2^{\frac{n}{2}}, 2^{\frac{\lambda}{2}}\})$ queries in the seedless robustness model, where n is the output size of the internal primitives and λ is the min-entropy of the entropy source. Our result implies 128-bit security given $n = 256$ and $\lambda = 256$ for Linux-DRBG. We also present two distinguishing attacks using $O(2^{\frac{n}{2}})$ and $O(2^{\frac{\lambda}{2}})$ queries, respectively, proving the tightness of our security bound.

Keywords: Deterministic random bit generator · Linux-DRBG · Seedless robustness · Provable security

1 Introduction

Deterministic Random Bit Generator. Producing random numbers plays a crucial role in cryptography, serving for the generation of secret keys, IVs and nonces (for encryption modes), and passwords (for identification protocols), to name a few. In practice, random bits are often generated using a deterministic random bit generator (DRBG), which refers to an algorithm that generates random bits using a seed value obtained from a physical source with a sufficient amount of entropy. The term "deterministic" means that there is no inherent randomness in the algorithm itself. DRBGs find applications in various environments such as simulation, encryption, etc.

Provable Security of DRBG. The randomness of the bits produced by a DRBG has typically been evaluated by statistical criteria. On the other hand, there have been attempts to prove the security of DRBGs through the provable security method in cryptography [4,6,16,20,22,25] as many constructions

This work was done while H. Kim and Y. Lee were PhD students at KAIST.

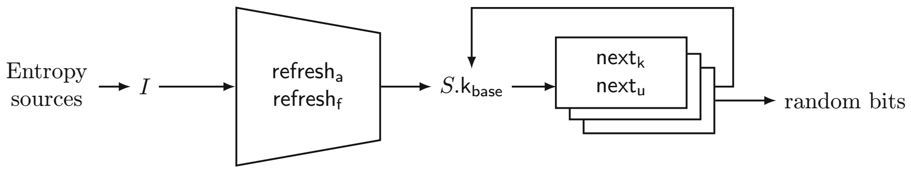


Fig. 1. Overall structure of Linux-DRBG. refresh$_a$ and refresh$_f$ are entropy absorbing functions, next and next_user are random bit generating functions.

are based on cryptographic primitives such as block ciphers and hash functions. As part of this effort, Dodis et al. [12] proposed a security model for DRBGs, demonstrated that a random bit generator used in the Linux operating system is not secure under the proposed model, and suggested its modification. In this paper, the security notions for DRBGs are distinguished as robustness, forward security, backward security, and resilience. The security model incorporates a hardware random bit generator used to generate seeds in DRBGs and a virtual system distribution sampler to model the hardware random bit generator and adversarial manipulation on it. Based on this model, the robustness of the sponge structure has been proved [13], and subsequently, the robustness of HMAC-DRBG and HASH-DRBG, both recommended by NIST.SP.800-90A, was proved [23]. Recently, CTR-DRBG, also recommended by NIST.SP.800-90A, has been proved [17].

Seedless Robustness Model. Dodis et al. [10] pointed out a limitation of the existing model, which assumes and exploits randomness called a *seed*, unknown to adversaries and kept secret. The assumption is not realistic in a practical scenario, and a DRBG in such a model cannot be considered deterministic since the seed implies the existence of randomness in addition to entropy. They proposed a seedless robustness model and demonstrated that CTR-DRBG is not secure under the new model. They also proposed new DRBGs that are secure under the seedless robustness model.

Research on Linux-DRBG. Linux is one of the widely-used computer operating systems developed as open-source software through collaborative efforts within the community. Linux utilizes DRBGs to generate random bits. The incorporation of DRBGs in Linux dates back to version 1.3.30 in 1994, and since then, modifications and enhancements have been ongoing. Barak et al. [2] suggested the robustness model and discussed the robustness of Linux-DRBG, and Gutterman et al. [15] introduced an attack on Linux 2.6.10 DRBG, and Linux has fixed the DRBG in following versions. Goichon et al. studied on entropy propagation in Linux- DRBG [14] in 2012. Dodis et al. [12] mentioned above, modified the robustness model, demonstrating that Linux-DRBG is not robust exploiting its entropy estimating process with timer randomness. This vulnerability has been fixed in subsequent versions of Linux by modifying Linux-DRBG to collect and estimate entropy from a variety of entropy sources. Linux-DRBG has been significantly modified in Linux versions 4.0 and 5.17[1]. However, the security of the updated Linux-DRBG has not yet been proven.

THE STRUCTURE OF LINUX-DRBG. The overall structure of Linux-DRBG in version 6.4.8 of Linux is shown in Fig. 1. Linux-DRBG collects and estimates entropy from a variety of entropy sources such as hardware, timers, interrupts, and bootloaders.

It then updates the base state with collected entropy, through entropy accumulating functions like procedure $\mathsf{refresh_a}$ and $\mathsf{refresh_f}$.

When a user runs random bit generating functions such as $\mathsf{next_k}$ and $\mathsf{next_u}$, the state utilizes one of the CPU core's states (CPU state) to update and then generate random bits. During this process, the base state is also re-updated.

Linux-DRBG utilizes two cryptographic primitives: for the entropy accumulating functions, $\mathsf{refresh_a}$ and $\mathsf{refresh_f}$, it uses the hash function BLAKE2s, and for the random bit generating functions, $\mathsf{next_k}$ and $\mathsf{next_u}$, it employs the stream cipher ChaCha20. The internal structures of BLAKE2s and ChaCha20 have been modeled by Luykx et al. [18] and Degabriele et al. [11], respectively.

1.1 Our Contribution

Since the significant structural changes in Linux 4 and Linux 5.17, there has been no research on the provable security of Linux-DRBG. For the first time (to the best of our knowledge), we formally model Linux-DRBG in Linux 6.4.8 and prove its security in the seedless robustness model.

According to the source code of Linux 6.4.8, Linux-DRBG has two entropy accumulating functions, $\mathsf{refresh_a}$ and $\mathsf{refresh_f}$, and two random bit generating functions, $\mathsf{next_k}$ and $\mathsf{next_u}$. We abstracted Linux-DRBG into the 4 functions and adjusted its structure that does not fit into the existing seedless robustness model. The process of analyzing the source code to abstract Linux-DRBG is detailed in Sect. 4.

We prove that Linux-DRBG is secure up to $O(\min\{2^{n/2}, 2^{\lambda/2}\})$ where n is the output size of the internal primitives and λ is the min-entropy of the entropy source. Since $n = 256$ and $\lambda = 256$ in Linux-DRBG, our security bound implies the 128-bit security of Linux-DRBG in the seedless robustness model. We also present two distinguishing attacks using $O(2^{\frac{n}{2}})$ and $O(2^{\frac{\lambda}{2}})$ queries, respectively, proving the tightness of our security bound.

Dodis et al. used the reducing technique(called game hopping) to prove robustness by splitting the security into r individual recovering security and preserving security, where r is the number of random bit generating query [12]. Subsequent papers proving the robustness of DRBGs, except for cases like the

[1] The modified Linux-DRBG was primarily designed and developed by Jason A. Donenfeld. See https://github.com/torvalds/linux/blob/master/drivers/char/random.c.

direct proof of CTR-DRBG's robustness in 2020 [17], have predominantly utilized this approach [13,23]. This methodology is also applied in the seedless robustness model [10]. However, applying this method directly to Linux-DRBG would only yield $n/3$-bit security. In this paper, as shown in Lemma 2, we have adopted a different game hopping technique to split Linux-DRBG's robustness. Through this methodology, we could prove that Linux-DRBG is secure up to $O(2^{n/2})$ adversarial queries. We believe that this proof method could be beneficial in proving the robustness of other DRBGs.

2 Preliminaries

We write 0^n to denote the n-bit string of all zeros. Given a non-empty finite set $\mathcal{X}$, $x \leftarrow_\$ \mathcal{X}$ denotes that x is chosen uniformly at random from $\mathcal{X}$. For a set $\mathcal{X}$, $|\mathcal{X}|$ denotes the number of elements in $\mathcal{X}$. The set of all permutations of $\{0,1\}^n$ is denoted $\mathsf{Perm}(n)$. For a keyed function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ with key space $\mathcal{K}$ and non-empty sets $\mathcal{X}$ and $\mathcal{Y}$, we will write $F_K(\cdot)$ to denote $F(K,\cdot)$ for $K \in \mathcal{K}$. For a set $S \subseteq \{0,1\}^n$ and $x \in \{0,1\}^n$, we write $S \oplus x$ to denote $\{s \oplus x : s \in S\}$. Let $\chi = (A,B,C)$ be a list. We write $\chi.\mathit{append}(D)$ to append an element to the χ. Thus, after appending, $\chi = (A,B,C,D)$. We denote $|$ as a bitwise OR operation.

For a (binary) string x, $|x|$ denotes the length of x. The empty string is denoted ε, where $|\varepsilon| = 0$. For an ℓ-bit string x, and m and n such that $0 \le m \le n \le \ell - 1$, $x[m:n]$ denotes an $(n-m+1)$-bit string from the m-th bit to the n-th bit of x, and $x[m:]$ denotes an $(\ell - m + 1)$-bit string from the m-th bit to the last bit of x. When $M = M_1 \| \cdots \| M_w$ where $|M_i| = t$ for $1 \le i \le w-1$ and $0 < |M_w| \le t$, we write $(M_1, \ldots, M_w) \xleftarrow{t} M$. For an integer $0 \le i < 2^s$, $\langle i \rangle_s$ denotes s-bit representation of i. For a real number t, $\lceil t \rceil$ is the smallest integer that is the same as or bigger than t. For $X \in \{0,1\}^n$, we define $\mathsf{msb}_m(X)$ (resp. $\mathsf{lsb}_m(X)$) as m most significant bits of X (resp. m least significant bits of X).

For a tuple $S_A = (X,Y,Z)$, we can access X inside S_A as $S_A.X$.

Random permutation. A random permutation is a bijective mapping from a finite set to itself, selected uniformly at random from all possible permutations of the set. We treat 20 rounds of a ChaCha20 cipher as a random permutation π which is selected from $\mathsf{Perm}(2n)$ [3,11].

Block ciphers. A block cipher, modeled as an *ideal cipher*, is a keyed function $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ where for a fixed key $K \in \{0,1\}^k$, $E_K(\cdot)$ is a random permutation that is uniformly chosen from $\mathsf{Perm}(n)$. For the rest of the paper, we let $\Pi(k,n)$ denote the set of all n-bit block ciphers using k-bit keys.

If a security proof supposes a block cipher is picked uniformly random from $\Pi(k,n)$ at the beginning of the query and allows the adversary to make queries to the block cipher, we call the proof is modeled under an *ideal cipher model.*

DRBG. From [10,12,21], a DRBG(Deterministic Random Bit Generator) is a triple of algorithms $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ where:

- setup: an algorithm that outputs an initial state S.

– refresh: an entropy accumulating algorithm that, given a state S and an input I, outputs a new state S'.
– next: a random bit generating algorithm that, given a state S, outputs a new state S' and random bits R.

However, Linux-DRBG does not fully fit in the above DRBG model, it has two refresh functions and two next functions. These functions are described in Algorithm 4 and we explained the reason that we modeled Linux-DRBG in this format in Sect. 4.

Distinguishing Game. Throughout this paper, we prove the robustness of Linux-DRBG by showing that Linux-DRBG and its subalgorithms are **indistinguishable** from an ideal DRBG with a distinguishing game. Generally, let $\mathcal{G}_0$ and $\mathcal{G}_1$ be algorithms and $\mathcal{A}$ be an adversary to distinguish them. Then the distinguishing game is composed as below.

1. $b \leftarrow_{\$} \{0, 1\}$, then $\mathcal{G}_b$ make interfaces that $\mathcal{A}$ can access and get response. The interfaces are called oracles.
2. Under the prescribed rule, $\mathcal{A}$ makes queries to the oracle and gets responses.
3. After querying phase, $\mathcal{A}$ outputs b'. If $b' = b$, $\mathcal{A}$ wins.

Let the distinguishing game between $\mathcal{G}_0$ and $\mathcal{G}_1$ be dist. Then the distinguishing advantage of $\mathcal{A}$ against dist, $\mathbf{Adv}_{\mathsf{dist}}(\mathcal{A})$ is defined as below.

$$\begin{aligned}\mathbf{Adv}_{\mathsf{dist}}(\mathcal{A}) &= |\Pr[1 \leftarrow \mathcal{A}|b = 0] - \Pr[1 \leftarrow \mathcal{A}|b = 1]| \\ &= |\Pr[0 \leftarrow \mathcal{A}|b = 0] - \Pr[0 \leftarrow \mathcal{A}|b = 1]| .\end{aligned}$$

Min Entropy. Let the prediction probability of a random variable X be

$$\mathsf{Pred}(X) := \max_x \Pr[X = x] .$$

Then for another random variable Y, $\mathsf{Pred}(X|y) := \max_x \Pr[X = x|Y = y]$. Then conditional probability of X over Y is

$$\mathsf{Pred}(X|Y) := E(\mathsf{Pred}(X|y)).$$

And (average-case) conditional min-entropy is

$$H_\infty(X|Y) = -\log(\mathsf{Pred}(X|Y)).$$

3 Seedless Robustness Model

The seedless robustness model [10] is a modification of the "seeded" robustness model [12], designed to relax the unrealistic assumption of the original model, namely, the existence of a random seed that should be kept secret to an adversary and independent of the entropy source.

Seedless Robustness Oracle. Let

$$\mathcal{G} = (\mathsf{setup}[P], \mathsf{refresh}[P], \mathsf{next}[P])$$

be a DRBG based on an ideal primitive P (such as a random oracle, an ideal cipher, or a random permutation), where P is chosen uniformly at random from the set of all possible primitives, denoted $\mathcal{P}$. Then the seedless robustness oracles are defined as described in Algorithm 1, where c denotes the entropy accumulated in the DRBG and λ is a fixed parameter (denoting the minimum required for the accumulated entropy).

Algorithm 1. Oracles for Seedless Robustness Game

```
   Procedure INIT()
1: $b \leftarrow_{\$} \{0,1\}$
2: $P \leftarrow_{\$} \mathcal{P}$, $c \leftarrow 0$
3: $S \leftarrow \mathsf{setup}[P]()$
4: return $P$

   Procedure REF$(I, \gamma)$
1: $S \leftarrow \mathsf{refresh}[P](S, I)$; $c \leftarrow c + \gamma$
2: return $\gamma$

   Procedure ROR$(len)$
1: $(S, y_1) \leftarrow \mathsf{next}[P](S, len)$
2: if $c < \lambda$ then
3:     $c \leftarrow 0$; return $y_1$
4: $y_0 \leftarrow_{\$} \{0,1\}^{len}$
5: return $y_b$

   Procedure GET();
1: $c \leftarrow 0$; return $S$

   Procedure SET$(S^*)$
1: $S \leftarrow S^*$; $c \leftarrow 0$

   Procedure $P(x)$
1: return $P(x)$

   Procedure $P^{-1}(x)$ //If exists
1: return $P^{-1}(x)$
```

Seedless adversary. In seedless robustness model [10], an adversary $\mathcal{A}$ consists of two algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$. The relationship between $\mathcal{A}_1$ and $\mathcal{A}_2$ is as follows:

- $\mathcal{A}_1$ is allowed to make queries only to the entropy accumulating oracle REF, while $\mathcal{A}_2$ is allowed to make queries to all the other oracles except REF,
- $\mathcal{A}_1$ knows all the queries made by $\mathcal{A}_2$ and the corresponding responses, while $\mathcal{A}_2$ observes only the responses to the queries made by $\mathcal{A}_1$ without knowing the queries themselves.

$\mathcal{A}_1$ is modeled in a way that the adversary can influence entropy accumulation but cannot ascertain specific values of entropy inputs. In the "seeded" robustness model [12], the distribution sampler $\mathcal{D}$ provides entropy inputs. However, as argued in [10], the security proof using $\mathcal{D}$ is based on an unrealistic assumption that $\mathcal{D}$ is independent of the underlying primitive of the DRBG (In Linux-DRBG, E, and π). Instead, they replaced the distribution sampler with $\mathcal{A}_1$ that only accumulates entropy in the DRBG.

To model the quality of the entropy source, we define legitimacy for $\mathcal{A}$. Let $\mathcal{I}_i$ be the random variable for i-th input $\mathcal{A}_1$ makes, and $\mathcal{T}_i$ be the random variable for all input-output list of robustness game, excluding I_i, the i-th entropy input. Then $\mathcal{A}$ is γ^***-legitimate** if

$$H_\infty(\mathcal{I}_i|\mathcal{T}_i) \geq \gamma_i \geq \gamma^*.$$

for every i. Against a γ^*-*legitimate* adversary $\mathcal{A}$, the seedless robust game is defined as follows.

Seedless Robustness Game.

1. INIT is executed.
2. $\mathcal{A}_1$ makes queries to REF, while $\mathcal{A}_2$ makes queries to ROR, GET, SET, P and P^{-1} (if available) in an interleaved manner.
3. $\mathcal{A}_2$ outputs $b' \in \{0,1\}$.
4. If $b' = b$, then $\mathcal{A}$ wins, and $\mathcal{A}$ loses otherwise.

The game begins with the procedure INIT, which chooses a random bit b and a random primitive P. It then runs `setup` to make the initial state. The other parts of the game are oracles offered to an adversary $\mathcal{A}$:

- REF: state S is updated by calling `refresh` with input I of entropy at least γ,
- ROR: state S is updated by calling `next`, and then y_0 and y_1 are prepared: if $c < \lambda$, then it returns y_1 regardless of b, while if $c \geq \lambda$, then y_b is returned
- GET: returns the state S of the DRBG.
- SET(S^*): sets the state S of the DRBG to S^*.
- P returns the result of a primitive query. If the primitive allows inverse query(i.e., block cipher), the oracle should also provide P^{-1}.

Now we can distinguish two worlds $\mathcal{G}_0$ and $\mathcal{G}_1$ from the seedless robust games according to the bit $b \in \{0,1\}$, and for any DRBG $\mathcal{G}$, the seedless robustness security of $\mathcal{G}$ against $\mathcal{A}$ is defined as follows.

$$\mathbf{Adv}^{\mathcal{G}}_{rob}(\mathcal{A}) \overset{\text{def}}{=} \mathbf{Adv}_{\mathsf{dist}}(\mathcal{A}).$$

4 Modeling Linux-DRBG

In this section, we model Linux-DRBG to fit into the Seedless Robustness security model. In this paper, we studied the Linux version 6.4.8.[2] We mainly

[2] https://github.com/torvalds/linux/blob/master/drivers/char/random.c .

analyzed a `random.c` file in Linux 6.4.8, a collection of functions related to Linux-DRBG. Linux-DRBG uses the hash function BLAKE2s and the stream cipher ChaCha20 as internal primitives. We first describe the modeling of the two internal primitives and then explain the overall structure of Linux-DRBG.

4.1 Underlying Primitives and Their Modeling

Algorithm 2. A Procedure in the BLAKE2s

```
   COMP : $\{0,1\}^{n/4} \times \{0,1\}^n \times \{0,1\}^* \to \{0,1\}^n$
   Procedure COMP$[E](t, h, I)$
1: $len \leftarrow |I|$
2: $rem \leftarrow len - 2n(\lceil len/2n \rceil - 1)$
3: $(I_1, \ldots, I_\ell) \xleftarrow{2n} I$
4: $I_\ell \leftarrow 0^{2n-rem} \| I_\ell$
5: $h_1 \leftarrow h$
6: for $i \leftarrow 1$ to $\ell - 1$ do
7:     $h_{i+1} \leftarrow B[E](h_i, t + i \cdot 2n, I_i)$
8: $y \leftarrow B'[E](h_\ell, t + len, I_\ell)$
9: return $y$
```

4.1.1 BLAKE2s

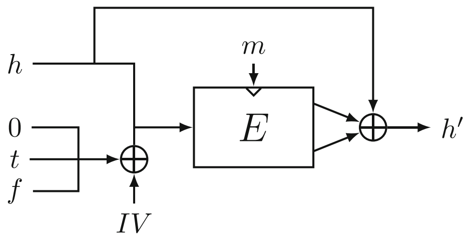
The Linux DRBG uses the BLAKE2s hash function to accumulate entropy [1]. Due to the known structure of the BLAKE2s primitive E, it cannot be modeled as an ideal cipher [5,7,18]. In this section, we first introduce an appropriate model for E, the mappable weak block cipher, and then describe the structure of BLAKE2s based on the model.

Weak Block Cipher. Consider a partition $\{0,1\}^n = \mathcal{W} \cup (\{0,1\}^n \backslash \mathcal{W})$. Define the set of weakly ideal ciphers $\Pi_w(k, n, \mathcal{K}, \mathcal{W})$ with a weak key set $\mathcal{K}$ as the collection of all $E \in \Pi(k, n)$ that satisfies the following properties: For every $K \in \mathcal{K}$, $W \in \mathcal{W}$, and $X \in \{0,1\}^n \backslash \mathcal{W}$

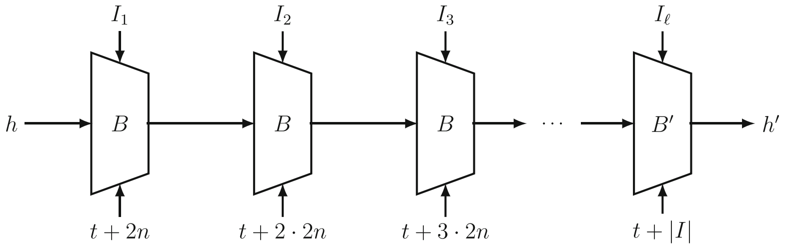
$$E_K(W) \in \mathcal{W},$$
$$E_K(X) \in \{0,1\}^n \backslash \mathcal{W}.$$

Similar to the ideal cipher model, if a security proof assumes a weak block cipher is picked uniformly random from $\Pi_w(k, n, \mathcal{K}, \mathcal{W})$ at the beginning of the query, we call the proof is under *weakly ideal cipher model.*

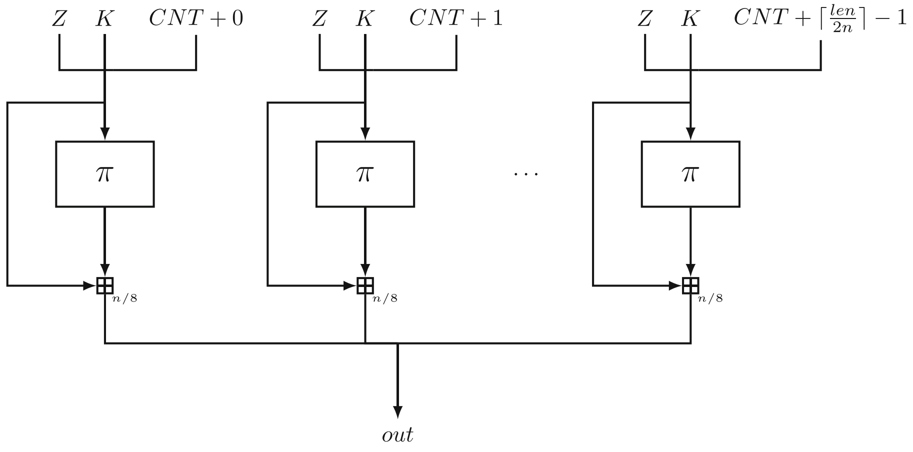
Mappable Weak Block Cipher. To model BLAKE2s, we extend the definition of a weak block cipher to a *mappable* weak block cipher. An example of



(a) 1 round of BLAKE2s compressing functions. $f = 0^{n/4}$ for B or $f = 0^{n/8} \parallel 1^{n/8}$ for B'.



(b) COMP structure



(c) CB structure

Fig. 2. BLAKE2s and ChaCha20 internal structure

a mappable block cipher, along with the reasoning behind its definition, is provided in the following subsection. Define a set of *mappable* weak block ciphers $\mathit{\Pi}_{mw}(k, n, \mathcal{K}, \mathcal{W}, f, \mathcal{W}_f)$ with a function f, a set $\mathcal{W}_f$, and $\mathit{\Pi}_w(k, n, \mathcal{K}, \mathcal{W})$ as the collection of all weak block cipher $E \in \mathit{\Pi}_w(k, n, \mathcal{K}, \mathcal{W})$ that satisfies the following property: For every $W \in \mathcal{W}$,

$$f(W) \in \mathcal{W}_f.$$

Similar to the weakly ideal cipher model, if a security proof supposes a mappable weak block cipher is picked uniformly random from $\Pi_{mw}(k, n, \mathcal{K}, \mathcal{W}, f, \mathcal{W}_f)$ at the beginning of the query, we call the proof is under *mappable weakly ideal cipher model.*

The Modeling of the primitive E. We model E as a mappable weak block cipher. For the BLAKE2s, a weak key set $\mathcal{K}$, a weak input set $\mathcal{W}$, $\mathcal{W}_{\mathsf{sum}}$, and $\mathsf{sum} : \mathcal{W} \to \mathcal{W}_{\mathsf{sum}}$ are defined as follows:

$$\begin{aligned}
\mathcal{W} &= \{aeaebfbfcgcgdhdh \in \{0,1\}^{2n} \mid a,b,c,d,e,f,g,h \in \{0,1\}^{n/8}\},\\
\mathcal{K} &= \{kkkkkkkkkkkkkkkk \in \{0,1\}^{2n} \mid k \in \{0,1\}^{n/8}\}\\
\mathcal{W}_{\mathsf{sum}} &= \{wxwxyzyz \in \{0,1\}^{n} \mid w,x,y,z \in \{0,1\}^{n/8}\},\\
\mathsf{sum}(W) &= \mathsf{msb}_n(W) \oplus \mathsf{lsb}_n(W).
\end{aligned}$$

Consequently, BLAKE2s' primitive E belongs to $\Pi_{mw}(2n, 2n, \mathcal{K}, \mathcal{W}, \mathsf{sum}, \mathcal{W}_{\mathsf{sum}})$, and we prove the security of Linux-DRBG under the mappable weakly ideal cipher model for $\Pi_{mw}(2n, 2n, \mathcal{K}, \mathcal{W}, \mathsf{sum}, \mathcal{W}_{\mathsf{sum}})$.

To model the properties of E [7], at least the weakly ideal cipher model is required. To prove the security of Linux-DRBG, we need to distinguish the output of E from a random number. Due to the initialization vector (IV) of BLAKE2s, it is impossible for E to receive an input that corresponds to $\mathcal{W}$. Since the compression function of BLAKE2s sums halves of E's output, we need to determine whether the random number falls within $\mathcal{W}_{\mathsf{sum}}$, not $\mathcal{W}$. Consequently, to use $\mathcal{W}_{\mathsf{sum}}$, we define and employ the *mappable* weakly ideal cipher model.

The modeling of the compression function. For $x \in \{0,1\}^{2n}$, let $\mathsf{TRSum}(x) = x[0 : n-1] \oplus x[n :]$. Then the 1 round of compression function of the BLAKE2s is defined as follows:

$$\begin{aligned}
B[E](h,t,I) &\leftarrow \mathsf{TRSum}(E(I, h \,\|\, (0^{n/2} \,\|\, t \,\|\, 0^{n/4}) \oplus IV)) \oplus h,\\
B'[E](h,t,I) &\leftarrow \mathsf{TRSum}(E(I, h \,\|\, (0^{n/2} \,\|\, t \,\|\, 0^{n/8} \,\|\, 1^{n/8}) \oplus IV)) \oplus h
\end{aligned}$$

where $E \in \Pi_{mw}(2n, 2n, \mathcal{K}, \mathcal{W}, \mathsf{sum}, \mathcal{W}_{\mathsf{sum}})$, h is a n-bit value, t is a $n/4$-bit counter, I is an $2n$-bit input, and $IV = IV_1 \| \cdots \| IV_8$ is n-bit fixed string where $IV_k \in \{0,1\}^{n/8}$ for all $1 \le k \le 8$ and $IV_i \ne IV_j$ for all $1 \le i \ne j \le 8$. $B[E](h,t,I)$ and $B'[E](h,t,I)$ are described in Fig. 2(a). Here, B is used when I is not the final input block, and B' is used when I is the final input block. We represent Linux-DRBG function `blake2s_compress` as COMP in Algorithm 2 and Fig. 2(b).

Avoiding the weak state. In BLAKE2s, the IV is a 256-bit string consisting of a tuple of 32-bit values derived from the square roots of distinct primes starting from 2 and ending at 19. Note that the square roots of primes are all different. Hence, in actual usage, the weak input $w \in \mathcal{W}$ of BLAKE2s cannot be accessed due to the distinct elements of IV. Therefore, we assume that the $IV = IV_1 \| \cdots \| IV_8$ is n-bit fixed string where $IV_k \in \{0,1\}^{n/8}$ for all $1 \le k \le 8$ and $IV_i \ne IV_j$ for all $1 \le i \ne j \le 8$.

Algorithm 3. A Procedure in the ChaCha20

```
   $\mathsf{CB} : \{0,1\}^n \times \{0,1\}^{n/2} \times \{0,1\}^* \rightarrow \{0,1\}^*$
   **Procedure** $\mathsf{CB}[\pi](K, CNT, len)$
1: $out \leftarrow \epsilon$
2: **while** $len > 0$ **do**
3:     $B \leftarrow \pi(Z \parallel K \parallel CNT) +_{n/8} (Z \parallel K \parallel CNT)$
4:     $CNT \leftarrow CNT + 1$
5:     $out \leftarrow out \parallel B$
6:     $len \leftarrow len - 2n$
7: **return** $out$
```

4.1.2 ChaCha20

Linux-DRBG produces pseudorandom outputs using ChaCha20. The stream cipher ChaCha20 is known to be faster than AES when hardware support for AES is not available. The ChaCha20 uses a fixed constant Z, expressed as the hexadecimal representation of "expand 32-byte k".

The random permutation model. The 20 rounds of the ChaCha20 can be modeled as a random permutation $\pi \leftarrow_\$ \mathsf{Perm}(2n)$ [11]. Using the random permutation π and a fixed constant Z, we model a function `chacha_block` as CB in the Algorithm 3 and Figure 2(c). The computation of CB on one $2n$ bits block is expressed as follows:

$$out \leftarrow \pi(Z \parallel K \parallel CNT) +_{n/8} (Z \parallel K \parallel CNT)$$

where, Z is the fixed $n/2$-bit constant, K is an n-bit key, and CNT is an $n/2$-bit counter, and $+_{n/8}$ represents word-by-word modulo $2^{n/8}$ addition.

4.2 Overall Structure of Linux-DRBG

We divide Linux-DRBG into three parts: Initialization, Entropy Accumulation, and Random Bit Generation. Linux-DRBG is initialized when the Linux kernel boots. Then, Linux-DRBG starts to accumulate entropy from various hardware sources. When accumulated entropy is larger than 256-bit, a character device file `/dev/random`, a system call `getrandom`, and a kernel interface `get_random_bytes` of Linux-DRBG can produce random bits. In this subsection, we describe the three parts of Linux-DRBG. We also describe how we model each part in the Seedless Robustness model which will be described in the Algorithm 4 and Fig. 3 later.

4.2.1 Initialization

When the Linux kernel boots, Linux-DRBG initializes a state `input_pool` of the BLAKE2s. Also, Linux-DRBG initializes states `base_crng` and `crng` of the ChaCha20. Then Linux-DRBG calls a function `random_init_early` that accumulates entropy in the `input_pool` without accessing the timer in Linux. Finally, when the timer becomes available, Linux-DRBG calls a function `random_init` which accumulates entropy in the `input_pool` using the timer in Linux.

The Modeling of the Initialization. We model the three states `input_pool`, `base_crng`, and `crng` in a single state S. Using the state S, we model the initialization as a function setup() in the Algorithm 4. Note that Linux-DRBG accumulates some entropy in the initialization. But we exclude entropy accumulation in the setup. Then, we model an attacker to call entropy accumulation functions with high entropy after entropy-draining events including setup.

4.2.2 Entropy Accumulation

Linux-DRBG accumulates entropy from various hardware entropy sources. Linux-DRBG calls functions to access hardware entropy sources. The functions related to entropy accumulation and estimation are listed as follows:

- `add_hwgenerator_randomness`,
- `add_bootloader_randomness`,
- `add_interrupt_randomness`,
- `add_timer_randomness`.

When Linux-DRBG calls the functions, they return a string that contains entropy which is called "entropy input". Note that these functions also return an estimation of the entropy within their entropy inputs. Linux-DRBG credits the estimation using a function `credit_init_bits`, enabling it to track how much entropy has been accumulated in the state of the DRBG. An analysis of Linux kernel version 5.18.1 by the German Federal Office for Information Security (BSI) shows that the Linux entropy sources and their entropy estimations satisfy their security criteria [19]. Hence, we assume that all entropy sources and estimations are functioning correctly.

Entropy accumulating functions. Linux-DRBG utilizes the function `mix_pool_bytes` to use the BLAKE2s for accumulating entropy into its state. Also, Linux-DRBG uses a function `crng_reseed` to convert the BLAKE2s' state into a key for a random bit generation. Entropy accumulation works as follows:

1. Linux-DRBG obtains an entropy input from the entropy sources.
2. Linux-DRBG passes the entropy input to the hash function BLAKE2s.
3. `mix_pool_bytes`: BLAKE2s compresses the entropy input to its element h of its state.
4. `crng_reseed`: If Linux-DRBG needs to generate random bits, then the BLAKE2s uses h to derive keys for the random bit generation.

The Modeling of the Entropy Accumulation. We put constants h, t, and key $\mathsf{k}_{\mathsf{next}}$ of the BLAKE2s in the DRBG state S. Also, we put ChaCha20 key $\mathsf{k}_{\mathsf{base}}$ in the state S. In the Algorithm 4, the two entropy accumulating functions `mix_pool_bytes` and `crng_reseed` are modeled as $\mathsf{refresh}_{\mathsf{a}}$ and $\mathsf{refresh}_{\mathsf{f}}$.

We depict the two functions in the Fig. 3(a) and (b). In the Fig. 3(a), the $\mathsf{refresh_a}$ uses an entropy input I to update h. In the Fig. 3(b), the $\mathsf{refresh_f}$ uses h and an entropy input I to generate $\mathsf{k_{next}}$ for later BLAKE2s calls and $\mathsf{k_{base}}$ for random bit generation.

4.2.3 Random Bit Generation

Linux-DRBG uses either a function `get_random_bytes` or a function `get_random_bytes_user` to utilize the ChaCha20 for generating random bits. Note that Linux uses two types of the ChaCha20 for a multi-core system. A `base_crng` obtains the key $\mathsf{k_{base}}$ from the BLAKE2s and produces ckey_{CPU}. A `crng`, within each CPU, generates random bits utilizing the CPU with a key ckey_{CPU}. Random Bit Generation works as follows:

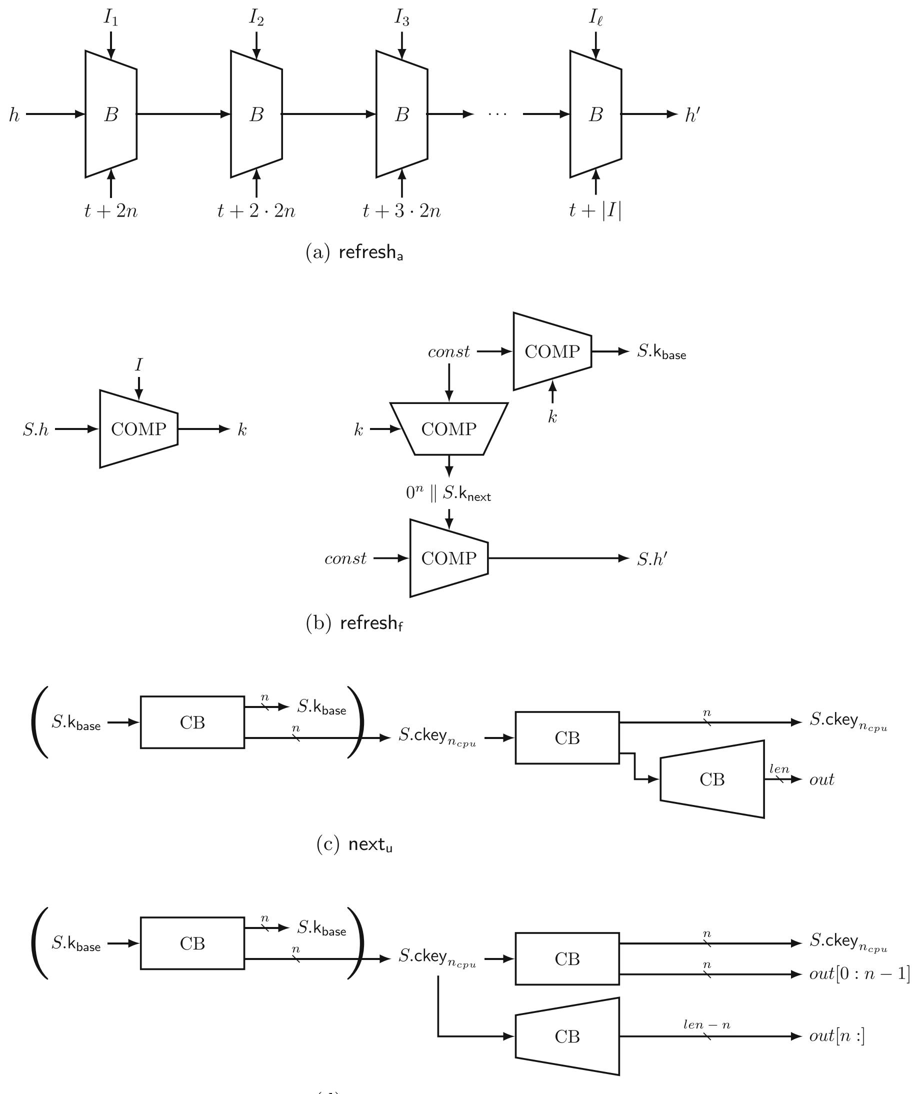
1. Linux-DRBG obtains the ChaCha20 key $\mathsf{k_{base}}$ from the BLAKE2s and initializes the `base_crng` using the $\mathsf{k_{base}}$.
2. Linux-DRBG assigns a CPU the random bit generation task.
   (a) If a flag $\mathsf{G\_flag}_{CPU}$ is set, Linux generates the CPU-specific key ckey_{CPU} from the `base_crng`.
   (b) Otherwise, Linux updates ckey_{CPU} by running `crng` with the old ckey_{CPU} without accessing $\mathsf{k_{base}}$.
3. Using the `crng` in the CPU, Linux-DRBG uses either the `get_random_bytes` or the `get_random_bytes_user` to generate random bits.

Two types of random bit generators. In Linux-DRBG, there are two types of random bit generation:

1. A first type generator produces random bits only if sufficient entropy is accumulated. The `/dev/random` is a representative example of the type.
2. A second type generator produces random bits at any time. The `/dev/urandom` is a representative example of the type.

The only difference between `/dev/random` and `/dev/urandom` is the fact that`/dev/random` prohibits *premature next* and `/dev/urandom` allows it. There already exists the study that any DRBG that allows *premature next* is insecure in the seedless model [9]. Therefore in this paper, we only consider the first type. The character device file `/dev/random`, the system call `getrandom`, and the kernel interface `get_random_bytes` are the first type generators. The `/dev/random` and the `getrandom` use the function `get_random_bytes_user` to generate random bits. The `get_random_bytes` kernel interface uses the function `get_random_bytes` to generate random bits. These generators can produce random bits only if accumulated entropy is more than 256-bit.

The Modeling of the Random Bit Generation. We put keys $\mathsf{k_{base}}$, ckey_1, $\ldots$, ckey_C, and flags $\mathsf{G\_flag}_1$, $\ldots$, $\mathsf{G\_flag}_C$ in the state S where C is the number of the CPUs. In the Algorithm 4, two random bit generation functions `get_random_bytes` and `get_random_bytes_user` are modeled as $\mathsf{next_k}$ amd $\mathsf{next_u}$. We depict the two functions in the Fig. 3(c) and (d).


I_1
I_2
I_3
I_ℓ
h
B
B
B
$\cdots$
B
h'
$t + 2n$
$t + 2 \cdot 2n$
$t + 3 \cdot 2n$
$t + |I|$
I
$S.h$
COMP
k
$const$
COMP
$S.\mathsf{k}_{\mathsf{base}}$
k
COMP
k
$0^n \parallel S.\mathsf{k}_{\mathsf{next}}$
$const$
COMP
$S.h'$
$S.\mathsf{k}_{\mathsf{base}}$
CB
n
$S.\mathsf{k}_{\mathsf{base}}$
n
$S.\mathsf{ckey}_{n_{cpu}}$
CB
n
$S.\mathsf{ckey}_{n_{cpu}}$
CB
len
out
$S.\mathsf{k}_{\mathsf{base}}$
CB
n
$S.\mathsf{k}_{\mathsf{base}}$
n
$S.\mathsf{ckey}_{n_{cpu}}$
CB
n
$S.\mathsf{ckey}_{n_{cpu}}$
n
$out[0 : n-1]$
CB
$len - n$
$out[n :]$

(a) $\mathsf{refresh_a}$

(b) $\mathsf{refresh_f}$

(c) $\mathsf{next_u}$

(d) $\mathsf{next_k}$

Fig. 3. Components of Linux-DRBG. States are defined in Sect. 4.3 and B, CB and COMP are defined in Sect. 4.1

4.3 Syntax of Linux-DRBG in Robustness Model

From the previous subsection, we establish the state of Linux-DRBG. Building upon this state, we present the syntax of Linux-DRBG, which constitutes our model of the operation of Linux-DRBG.

Algorithm 4. Syntax of Linux-DRBG

```
    setup : $\epsilon \rightarrow \mathcal{S}$
    Procedure setup()
 1: $S.h \leftarrow const$; $S.t \leftarrow 0$
 2: return $S$

    refresh_a : $\mathcal{S} \times (\{0,1\}^{2n})^* \rightarrow \mathcal{S}$
    Procedure refresh_a$[E](S, I)$
 1: if $S.\mathsf{k_{next}} \neq \epsilon$ then
 2:     $I \leftarrow 0^n \parallel S.\mathsf{k_{next}} \parallel I$
 3:     $S.\mathsf{k_{next}} \leftarrow \epsilon$
 4: $(I_1, \ldots, I_\ell) \xleftarrow{2n} I$
 5: for $i \leftarrow 1$ to $\ell$ do
 6:     $S.h \leftarrow B(S.h, S.t + i \cdot 2n, I_i)$
 7: $S.t \leftarrow S.t + |I|$
 8: return $S$

    refresh_f : $\mathcal{S} \times \{0,1\}^* \times \{0,1\}^n \rightarrow \mathcal{S}$
    Procedure refresh_f$[E](S, I, I_{cpu})$
 1: if $S.\mathsf{k_{next}} \neq \epsilon$ then
 2:     $I \leftarrow 0^n \parallel S.\mathsf{k_{next}} \parallel I$
 3: $k \leftarrow \mathrm{COMP}[E](S.t, S.h, I)$
 4: $I_{cpu} \leftarrow 0^n \parallel k \parallel I_{cpu} \parallel 0^{c-1}$
 5: $S.\mathsf{k_{next}} \leftarrow \mathrm{COMP}[E](0, const, I_{cpu} \parallel 0)$
 6: $S.\mathsf{k_{base}} \leftarrow \mathrm{COMP}[E](0, const, I_{cpu} \parallel 1)$
 7: $S.h \leftarrow const$; $S.t \leftarrow 2n$
 8: $S.\mathsf{G\_flag}_1 \leftarrow 1$; $S.\mathsf{G\_flag}_2 \leftarrow 1$; $\cdots$; $S.\mathsf{G\_flag}_C \leftarrow 1$
 9: return $S$

    next_k : $\mathcal{S} \times \{0,1\}^* \times \{0,1\}^{\log(C)} \rightarrow \mathcal{S} \times \{0,1\}^*$
    Procedure next_k$[\pi](S, len, n_{cpu})$
 1: if $S.\mathsf{G\_flag}_{n_{cpu}} = 1$ then
 2:     $tmp \leftarrow \mathsf{CB}[\pi](S.\mathsf{k_{base}}, 0, 2n)$
 3:     $S.\mathsf{k_{base}} \parallel S.\mathsf{ckey}_{n_{cpu}} \leftarrow tmp$
 4:     $S.\mathsf{G\_flag}_{n_{cpu}} \leftarrow 0$
 5: $k \leftarrow S.\mathsf{ckey}_{n_{cpu}}$
 6: $tmp \leftarrow \mathsf{CB}[\pi](S.\mathsf{ckey}_{n_{cpu}}, 0, 2n)$
 7: $S.\mathsf{ckey}_{n_{cpu}} \parallel out \leftarrow tmp$
 8: $B \leftarrow \mathsf{CB}[\pi](k, 1, len - n)$
 9: $out \leftarrow (out \parallel B)[0 : len - 1]$
10: return $(S, out)$

    next_u : $\mathcal{S} \times \{0,1\}^* \times \{0,1\}^{\log(C)} \rightarrow \mathcal{S} \times \{0,1\}^*$
    Procedure next_u$[\pi](S, len, n_{cpu})$
 1: if $S.\mathsf{G\_flag}_{n_{cpu}} = 1$ then
 2:     $tmp \leftarrow \mathsf{CB}[\pi](S.\mathsf{k_{base}}, 0, 2n)$
 3:     $S.\mathsf{k_{base}} \parallel S.\mathsf{ckey}_{n_{cpu}} \leftarrow tmp$
 4:     $S.\mathsf{G\_flag}_{n_{cpu}} \leftarrow 0$
 5: $tmp \leftarrow \mathsf{CB}[\pi](S.\mathsf{ckey}_{n_{cpu}}, 0, 2n)$
 6: $S.\mathsf{ckey}_{n_{cpu}} \parallel k \leftarrow tmp$
 7: if $len \leq n$ then
 8:     return $(S, k[0 : len - 1])$
 9: $out \leftarrow (\mathsf{CB}[\pi](k, 1, len))[0 : len - 1]$
10: return $(S, out)$
```

4.3.1 Internal State

We define the internal state of Linux-DRBG S as follows:

$$S := (h, t, \mathsf{k_{next}}, \mathsf{k_{base}}, \mathsf{ckey}_1, \cdots, \mathsf{ckey}_C, \mathsf{G\_flag}_1, \cdots, \mathsf{G\_flag}_C)$$

where C is the number of available CPUs. A description of each element in the state S is as follows:

- $h \in \{0,1\}^n$: a value that is updated by the compression function of the BLAKE2s,
- $t \in \{0,1\}^{n/4}$: a counter of the BLAKE2s,
- $\mathsf{k_{next}} \in \{0,1\}^n$: a key of the BLAKE2s,
- $\mathsf{k_{base}} \in \{0,1\}^n$: a key of the ChaCha20 that is used to produce CPU-specific keys ckey_1, ..., ckey_C,
- $\mathsf{ckey}_i \in \{0,1\}^n$: a key of the ChaCha20 that is used to produce random bits in the i-th CPU,
- $\mathsf{G\_flag}_i \in \{0,1\}$: a flag that indicates whether ckey_i needs to be updated by using $\mathsf{k_{base}}$ or not.

We define a set of states as

$$\mathcal{S} = \{0,1\}^n \times \{0,1\}^{n/4} \times \{0,1\}^n \times \{0,1\}^n \times (\{0,1\}^n)^C \times (\{0,1\})^C.$$

Modeling the BLAKE2s state. The compression function of BLAKE2s updates $S.h \in \{0,1\}^n$, and $S.t \in \{0,1\}^{n/4}$ serves as an input to BLAKE2s, accumulating the bit length of the input compressed. In Linux, BLAKE2s maintains a buffer and finalization flag in its state. The value to be compressed is stored in the buffer, and when Linux-DRBG reseeds, it initializes BLAKE2s' state and places a key in the buffer. We eliminated the buffer from S and stored the BLAKE2s key in $S.\mathsf{k_{next}}$. By saving $S.\mathsf{k_{next}}$, we can simulate BLAKE2s without the buffer. For the finalization flag in Linux, we explicitly incorporate it into the COMP in the Algorithm 2.

Modeling the ChaCha20 state. The final output of the BLAKE2s serves as the key for the ChaCha20, denoted as $S.\mathsf{k_{base}} \in \{0,1\}^n$. To leverage a multicore system, each CPU has its ChaCha20 states. Their keys are stored in $\mathsf{ckey}_1, \ldots, \mathsf{ckey}_C \in \{0,1\}^n$, where C represents the maximum available CPU number. When Linux-DRBG is asked to produce pseudorandom outputs, it determines whether the i-th CPU ChaCha20's key needs to be updated by using $S.\mathsf{k_{base}}$ or not based on $\mathsf{G\_flag}_1, \ldots, \mathsf{G\_flag}_C \in \{0,1\}$. If $\mathsf{G\_flag}_i = 1$, then it needs to be updated using $S.\mathsf{k_{base}}$. Otherwise, it is updated using old $S.\mathsf{ckey}_i$.

4.3.2 Linux-DRBG Syntax

We align Linux-DRBG with the syntax of the PRNG used in the robustness model [10,12]. The syntax of Linux-DRBG is detailed in Algorithm 4. Additionally, we illustrate the operations of each function in Fig. 3.

- $\mathsf{setup}()$: This algorithm produces an initial Linux-DRBG state S.

- $\mathsf{refresh_a}[E](S, I)$: Given a mappable weakly ideal cipher E, the state S and a variable length entropy input I, $\mathsf{refresh_a}$ compresses the entropy input I and store it to $S.h$.
- $\mathsf{refresh_f}[E](S, I, I_{cpu})$: Given a mappable weakly ideal cipher E, the state S, a variable length entropy input I, and a fixed n-bit input I_{cpu}, $\mathsf{refresh_f}$ compresses I and I_{cpu} to generate two keys $S.\mathsf{k_{next}}$ and $S.\mathsf{k_{base}}$. Note that $\mathsf{refresh_f}$ uses fixed constant *const* for compression. The *const* is defined as follows:

$$const \leftarrow IV;$$
$$const[0 : n/8-1] \leftarrow const[0 : n/8-1] \oplus (0^7 \parallel 1 \parallel 0^7 \parallel 1 \parallel 0^{n/8-16} | n << 8 | n).$$

  where the IV is the fixed constant from the BLAKE2s, and $|$ is the bit-wise OR operation. $S.\mathsf{k_{next}}$ is used in later calls to $\mathsf{refresh_a}$ or $\mathsf{refresh_f}$. $S.\mathsf{k_{base}}$ is used to generate random bits through $\mathsf{next_k}$ or $\mathsf{next_u}$. The $\mathsf{refresh_f}$ also sets flags $S.\mathsf{G\_flag}_1, \ldots, S.\mathsf{G\_flag}_C$.
- $\mathsf{next_k}[\pi](S, len, n_{cpu})$: Given a random permutation π, the state S, a required output length len, and a CPU number n_{cpu}, $\mathsf{next_k}$ generates len-bit random bits using $S.\mathsf{ckey}_{n_{cpu}}$ in the n_{cpu}-th CPU. If a $S.\mathsf{G\_flag}_{n_{cpu}}$ is set, then $\mathsf{next_k}$ updates $S.\mathsf{ckey}_{n_{cpu}}$ by using $S.\mathsf{k_{base}}$. Otherwise, $\mathsf{next_k}$ updates $S.\mathsf{ckey}_{n_{cpu}}$ by its old $S.\mathsf{ckey}_{n_{cpu}}$.
- $\mathsf{next_u}[\pi](S, len, n_{cpu})$: This algorithm works similar to the $\mathsf{next_k}$. The $\mathsf{next_k}$ directly uses $S.\mathsf{ckey}_{n_{cpu}}$ to produce random bits. But the $\mathsf{next_u}$ first produces a temporary key k, then it produces random bits using the k.

5 Robustness Proof

5.1 Linux-DRBG Robustness Game

Difference from the general DRBG model. From procedures in Algorithm 4, we can define robustness oracles for Linux-DRBG in Algorithm 5. Note that there are several differences from oracles in Algorithm 1 as follows.

- The refresh oracle is divided into $\mathrm{REF_a}$ and $\mathrm{REF_f}$ since Linux-DRBG accumulates the entropy gradually.
- If $\mathrm{REF_f}$ is invoked without sufficient entropy, it sets c to 0. Therefore, it is essential to supply sufficient entropy in a single $\mathrm{REF_f}$ call. This assumption is substantiated by the observation that, after system boot, Linux-DRBG invokes `crng_reseed` (equivalent to $\mathrm{REF_f}$) only if sufficient entropy is accumulated in its state. Also, after sufficient entropy is accumulated, Linux invokes `crng_reseed` every 60 s. After a sufficient amount of time has passed since the Linux system booted, it can be considered that 60 s is sufficient for accumulating enough entropy. Considering this behavior of Linux, even when a Seedless adversary attempts to leak the state of the DRBG, Linux-DRBG can be assumed to accumulate sufficient entropy with a single invocation of $\mathrm{REF_f}$. Therefore, $\mathrm{REF_f}$ can be modeled always to receive inputs with sufficient entropy.

Algorithm 5. Oracles for Linux-DRBG Seedless Robustness Game

```
    Procedure INIT()
1: $b \leftarrow_\$ \{0,1\}$, $c \leftarrow 0$
2: $E \leftarrow_\$ \Pi_{mw}(2n, 2n, \mathcal{K}, \mathcal{W}, \mathsf{sum}, \mathcal{W}_{\mathsf{sum}})$, $\pi \leftarrow_\$ \mathsf{Perm}(2n)$
3: $S \leftarrow \mathsf{setup}()$
4: return $(E, \pi)$

    Procedure $\mathrm{REF_a}[E](I, \gamma)$
1: $S \leftarrow \mathsf{refresh_a}[E](S, I)$; $c \leftarrow c + \gamma$
2: return $\gamma$

    Procedure $\mathrm{REF_f}[E](I, \gamma, I_{cpu})$
1: $S \leftarrow \mathsf{refresh_f}[E](S, I, I_{cpu})$
2: $c \leftarrow c + \gamma$
3: if $c \geq \lambda$ then
4:     $\mathsf{ready} \leftarrow 1$
5: else
6:     $c \leftarrow 0$
7: return $\gamma$

    Procedure $\mathrm{ROR_k}[\pi](len, n_{cpu})$
1: $(S, y_1) \leftarrow \mathsf{next_k}[\pi](S, len, n_{cpu})$
2: if $c < \lambda$ or $\mathsf{ready} = 0$ then
3:     $c \leftarrow 0$; $\mathsf{ready} \leftarrow 0$; return $y_1$
4: $y_0 \leftarrow_\$ \{0,1\}^{len}$
5: return $y_b$

    Procedure $\mathrm{ROR_u}[\pi](len, n_{cpu})$
1: $(S, y_1) \leftarrow \mathsf{next_u}[\pi](S, len, n_{cpu})$
2: if $c < \lambda$ or $\mathsf{ready} = 0$ then
3:     $c \leftarrow 0$; $\mathsf{ready} \leftarrow 0$; return $y_1$
4: $y_0 \leftarrow_\$ \{0,1\}^{len}$
5: return $y_b$

    Procedure GET();
1: $c \leftarrow 0$; $\mathsf{ready} \leftarrow 0$; return $S$

    Procedure SET($S^*$)
1: $S \leftarrow S^*$; $c \leftarrow 0$; $\mathsf{ready} \leftarrow 0$

    Procedure $E(k, x)$
1: return $E(k, x)$

    Procedure $E^{-1}(k, y)$
1: return $E^{-1}(k, y)$

    Procedure $\pi(x)$
1: return $\pi(x)$

    Procedure $\pi^{-1}(y)$
1: return $\pi^{-1}(y)$
```

- There are two ROR oracles, $\mathrm{ROR_k}$ and $\mathrm{ROR_u}$.
- The ROR oracles do not work correctly if $\mathsf{ready} = 0$, which means at least one $\mathrm{REF_f}$ call after entropy drain(will be defined in this section) is required.
- $\mathrm{ROR_k}$ and $\mathrm{ROR_u}$ requires the number of CPU to generate random bits, and the process varies whether $S.\mathsf{G\_flag}_{n_{cpu}}$ is 0 or 1.
- Since Linux-DRBG uses two cryptographic primitives, the mappable weakly ideal cipher E and the random permutation π, and they allow inverse query, there exist four primitive query oracles for Linux-DRBG. E. E^{-1}, π, π^{-1} are that.

With the above definition, the procedure for Linux-DRBG Robust Game is described as follows.

Linux-DRBG Robustness Game.

1. Oracle runs INIT() procedure.
2. Adversary $\mathcal{A}_2$ queries ROR_k, ROR_u, GET, SET, E, E^{-1}, π and π^{-1}. $\mathcal{A}_1$ queries REF_a and REF_f. All query orders are free and can be done multiple times.
3. $\mathcal{A}_2$ outputs $b' \in \{0,1\}$, if $b' = b$, $\mathcal{A}$ wins.

Entropy Drain. We define entropy drains, which are events that make the DRBG lose its entropy by giving some information to the adversary. The following events are called **entropy drains**:

– Exactly after INIT,
– Calling oracles to GET or SET,
– Calling an oracle ROR_k or ROR_u when $c < \lambda$ or $\mathsf{ready} = 0$.

For convenient notation, we denote an entropy drain as ED from now on.

Canonical adversary: An adversary $\mathcal{A}$ is called **canonical** if it follow the conditions below:

1. $\mathcal{A}_2$ queries ROR_k or ROR_u only when $c \geq \lambda$ and $\mathsf{ready} = 1$.
2. $\mathcal{A}_2$ queries ROR_k or ROR_u, only when the last construction query made by $\mathcal{A}_1$ is REF_f.
3. $\mathcal{A}_1$ does not query REF_a consecutively.
4. Between the last entropy drain and the first ROR_k or ROR_u query thereafter, $\mathcal{A}_2$ does not query GET and SET in situations where $c > 0$.
5. Between the last entropy drain and the first ROR_k or ROR_u query thereafter, $\mathcal{A}_1$ does not query REF_a.

We claim that we can assume the Linux-DRBG Robustness adversary $\mathcal{A}$ is canonical. This assumption comes from the fact that the only difference between $b = 0$ and $b = 1$ is in ROR_k or ROR_u when $c \geq \lambda$ and $\mathsf{ready} = 1$, and REF_a only accumulate entropy, and REF_f is required to transfer the accumulated entropy to $S.\mathsf{k}_\mathsf{base}$, the state used in ROR_k or ROR_u. Therefore, for any adversary $\mathcal{A}$ that violates the above condition, one can construct canonical adversary $\mathcal{A}'$ using $\mathcal{A}$ holding or simulating queries made by $\mathcal{A}$ appropriately. The strategy of $\mathcal{A}'$ is like below.

– If $\mathcal{A}$ violates condition 1 or 4, $\mathcal{A}'$ can easily simulate the query with primitive queries, because in that case the ideal world and real world behaviors are same.
– If $\mathcal{A}$ violates one of condition 2,3,5, $\mathcal{A}'$ just simply store REF_a queries after the last REF_f query. Then when $\mathcal{A}$ queries GET or REF_f, $\mathcal{A}'$ can concatenate the queries into a REF_a or REF_f.

Therefore it is reasonable to assume the Linux-DRBG Robustness adversary $\mathcal{A}$ is canonical.

5.2 Robustness Proof

The robustness advantage of Linux-DRBG is upper bounded in Theorem 1. In the statement of Theorem 1, REF (resp. ROR) calls mean REF_f and REF_a (resp. ROR_k and ROR_u) calls.

Theorem 1. *Let $\mathcal{A}$ be a λ-legitimate robustness game adversary that makes p primitive query, q_1* REF *query, q_2* ROR *query, ℓ_1 maximum number of entropy input block in a single* REF *call, ℓ_2 maximum number of output block in a single* ROR *call, σ_1 total number of entropy input blocks in every* REF*, and σ_2 total number of output blocks in every* ROR*. Let $\mathbf{Adv}_{rob}(p, q_1, q_2, \ell_1, \ell_2, \sigma_1, \sigma_2, \lambda)$ be the advantage upper bound of all possible adversaries $\mathcal{A}$. If $p+3q_1+2\sigma_1 \le 2^{n-1}$, the following inequality holds.*

$$\begin{aligned}
&\mathbf{Adv}_{rob}(p, q_1, q_2, \ell_1, \ell_2, \sigma_1, \sigma_2, \lambda)\\
&\le \frac{42q_1}{2^{0.5n}} + \frac{8q_2\ell_2(p+2q_2+\ell_2+\sigma_2)}{2^{2n}} + \frac{8q_1(p+3q_1+\sigma_1)}{2^\lambda}\\
&+ \frac{2p(p+8q_2+27q_1+2\sigma_1)+2q_1(72q_1+2\ell_1+31\sigma_1+2)+4q_2(8q_2+4\sigma_2+1)+4\sigma_1^2}{2^n}\\
&\le \frac{42\sigma_1}{2^{0.5n}} + \frac{8\sigma_2^2(p+4\sigma_2)}{2^{2n}} + \frac{8\sigma_1(p+4\sigma_1)}{2^\lambda}\\
&+ \frac{2p^2+2\sigma_1(29p+107\sigma_1+2)+4\sigma_2(4p+12\sigma_2+1)}{2^n}.
\end{aligned}$$

Let S_0 (resp. S_1) be a system of ideal (resp. real) world robustness oracles. In the INIT in Algorithm 5, if $b=0$ (resp. $b=1$), then the system of oracles is S_0 (resp. S_1). Note that the only differences between S_0 and S_1 are the return values of oracles ROR_k and ROR_u. If in S_0 (resp. S_1), they return y_0 (resp. y_1) when $c \ge \lambda$ and $\mathsf{ready} = 1$.

Methodology of the proof. Our proof involves dividing the robustness distinguishing game into subgames, proving the security of each, and then combining them. The subgames consist of the M-EXT game, which describes the distinguishing game for REF calls, the pREF_a game, the pREF_f game, and the distinguishing games for ROR calls, which include the bROR_k game, bROR_u, cROR_k game, and cROR_u game. In the text, we first define the hybrid world S_h for convenience of proof, ensuring that the state updates uniformly randomly when accumulated entropy $c \ge \lambda$ [17]. Subsequently, we define each subgame and its adversarial advantages, then claim Lemma 2 through game hopping with intermediate worlds that can apply each subgame, and prove the security of each subgame to prove Theorem 1 ultimately.

Among the subgames, the M-EXT game allows multiple M-EXT calls, differing from other subgames and previous proof methods that divided robustness into recovering security and preserving security [10,12,13,24]. Using the traditional method of splitting into several games with a single M-EXT call would result in each game's advantage having a $p^2/2^n$ term, and when gathering these, a $p^2q_1/2^n$ term would emerge, and we only could prove $O(2^{n/3})$ security for Linux-DRBG. In contrast, the M-EXT game, by allowing multiple M-EXT calls, eliminates the need to gather security bound, leading to $O(2^{n/2})$ security as shown in (2), and ultimately, we could prove that Linux-DRBG is secure up to $O(\min(2^{n/2}, 2^{\lambda/2}))$ adversarial queries. We believe this technique could be applied to other DRBGs as well, potentially helping to raise their security upper bounds.

Algorithm 6. Oracles for the hybrid world

```
Procedure $\mathrm{REF_a}^*(I, \gamma)$
1: $c \leftarrow c + \gamma$ //Update $c$ first.
2: if $c < \lambda$ then
3:     $S \leftarrow \mathsf{refresh_a}[E](S, I)$
4: else
5:     $S.h \leftarrow_\$ \{0,1\}^n$
6:     $S.t \leftarrow S.t + len$
7: return $\gamma$
```

```
Procedure $\mathrm{REF_f}^*(I, \gamma, I_{cpu})$
1: $c \leftarrow c + \gamma$ //Update $c$ first.
2: if $c < \lambda$ then
3:     $S \leftarrow \mathsf{refresh_f}[E](S, I, I_{cpu})$
4: else
5:     $\mathsf{k_{next}} \leftarrow_\$ \{0,1\}^n$
6:     $S.\mathsf{k_{base}} \leftarrow_\$ \{0,1\}^n$
7:     $S.h \leftarrow const$
8:     $S.t \leftarrow 2n$
9:     $\mathsf{ready} \leftarrow 1$
10:    $S.\mathsf{G\_flag}_1 \leftarrow 1$; $S.\mathsf{G\_flag}_2 \leftarrow 1$; $\cdots$; $S.\mathsf{G\_flag}_C \leftarrow 1$
11: return $\gamma$
```

```
Procedure $\mathrm{ROR_k}^*[\pi](len, n_{cpu})$
1: if $c < \lambda$ or $\mathsf{ready} = 0$ then
2:     $(S, y) \leftarrow \mathsf{next_k}[\pi](S, len, n_{cpu})$
3:     $c \leftarrow 0$; $\mathsf{ready} \leftarrow 0$
4: else
5:     if $S.\mathsf{G\_flag}_{n_{cpu}} = 1$ then
6:         $S.\mathsf{k_{base}} \leftarrow_\$ \{0,1\}^n$
7:         $S.\mathsf{G\_flag}_{n_{cpu}} \leftarrow 0$
8:     $S.\mathsf{ckey}_{n_{cpu}} \| y \leftarrow_\$ \{0,1\}^{n+len}$
9: return $y$
```

```
Procedure $\mathrm{ROR_u}^*[\pi](len, n_{cpu})$
1: if $c < \lambda$ or $\mathsf{ready} = 0$ then
2:     $(S, y) \leftarrow \mathsf{next_u}[\pi](S, len, n_{cpu})$
3:     $c \leftarrow 0$; $\mathsf{ready} \leftarrow 0$
4: else
5:     if $S.\mathsf{G\_flag}_{n_{cpu}} = 1$ then
6:         $S.\mathsf{k_{base}} \leftarrow_\$ \{0,1\}^n$
7:         $S.\mathsf{G\_flag}_{n_{cpu}} \leftarrow 0$
8:     $S.\mathsf{ckey}_{n_{cpu}} \| y \leftarrow_\$ \{0,1\}^{n+len}$
9: return $y$
```

We denote $\mathit{\Delta}_{\mathcal{A}}(S_0, S_1)$ for a seedless robustness distinguishing advantage of S_0 and S_1 for an adversary $\mathcal{A}$ satisfying conditions in Theorem 1. Let S_h be a system of hybrid words that contains oracle $\mathrm{REF_a}^*$, $\mathrm{REF_f}^*$, $\mathrm{ROR_k}^*$ and $\mathrm{ROR_u}^*$ in Algorithm 6 instead of $\mathrm{REF_a}$, $\mathrm{REF_f}$, $\mathrm{ROR_k}$ and $\mathrm{ROR_u}$ in Algorithm 5. That means, when $c \geq \lambda$, S_0 outputs bit outputs randomly but updates its state with Linux-DRBG algotirhm, and S_1 outputs bit outputs and updates its state with Linux-DRBG algotirhm, S_h outputs bit outputs and updates its state randomly. Then, by the triangle inequality, the following holds:

$$\Delta_{\mathcal{A}}(S_0, S_1) \leq \Delta_{\mathcal{A}}(S_0, S_h) + \Delta_{\mathcal{A}}(S_h, S_1). \quad (1)$$

With (1), the following lemma holds.

Lemma 1. *In a distinguishing game, $b \in \{0,1\}$ is uniformly randomly chosen to select one of two worlds. For $\Delta_{\mathcal{A}}(S_h, S_0)$, if $b = 0$ then S_h is selected. Otherwise, S_0 is selected. For $\Delta_{\mathcal{A}'}(S_h, S_1)$, if $b = 0$ then S_h is selected. Otherwise, S_1 is selected. Then for any robustness adversary $\mathcal{A}$, there exists $\mathcal{A}'$ that satisfies the following.*

$$\Delta_{\mathcal{A}}(S_h, S_0) \leq \Delta_{\mathcal{A}'}(S_h, S_1).$$

Proof. We can construct a distinguishing adversary $\mathcal{A}'$ between S_h and S_1 using an S_0 and S_h distinguishing adversary $\mathcal{A}$ as a subalgorithm. $\mathcal{A}'$ passes oracle queries of $\mathcal{A}$ to its oracles and just returns results to $\mathcal{A}$ except when $\mathcal{A}$ queries $\mathrm{ROR_k}$ or $\mathrm{ROR_u}$ and conditions $c \geq \lambda$ and $\mathsf{ready} = 1$ hold. If $\mathcal{A}$ queries $\mathrm{ROR_k}$ or $\mathrm{ROR_u}$ and conditions $c \geq \lambda$ and $\mathsf{ready} = 1$ hold, then $\mathcal{A}'$ randomly picks a bitstring and returns it to $\mathcal{A}$. If $b = 0$, then $\mathcal{A}'$ perfectly simulates S_h to $\mathcal{A}$, and if $b = 1$, then $\mathcal{A}'$ perfectly simulates S_0 to $\mathcal{A}$. Finally, $\mathcal{A}'$ outputs b', which is the final output of $\mathcal{A}$.

Hence, the following holds:

$$\Delta_{\mathcal{A}}(S_0, S_1) \leq 2\Delta_{\mathcal{A}'}(S_h, S_1).$$

Therefore, we only need to upper bound $\Delta_{\mathcal{A}'}(S_h, S_1)$.

5.2.1 Games for Robustness Proof

To upper bound $\Delta_{\mathcal{A}'}(S_h, S_1)$, we substitute oracles used in S_1 to oracles used in S_h using the game hopping technique. We employ subgames for each substitution. The subgames are M-EXT game, $\mathrm{pREF_a}$ game, $\mathrm{pREF_f}$ game, $\mathrm{bROR_k}$ game, $\mathrm{bROR_u}$ game, $\mathrm{cROR_k}$ game, $\mathrm{cROR_u}$ game. The former 3 sub games are necessary to substitute REF_a or REF_f with REF^*_a or REF^*_f, the latter 4 sub games are necessary to substitute $\mathrm{ROR_k}$ or $\mathrm{ROR_u}$ with $\mathrm{ROR_k}$ or $\mathrm{ROR_u}^*$. One can see how the following subgames are used to prove Theorem 1 via Lemma 2.

With oracles in Algorithm 7, M-EXT game, $\mathrm{pREF_a}$ game, $\mathrm{pREF_f}$ game processes are defined like below.

M-EXT **game.**

1. Oracle runs INIT() procedure.
2. Adversary $\mathcal{A}_2$ queries E, E^{-1} and $\mathcal{A}_1$ queries M-EXT$[E](inc, h_1, I, I_{\ cpu})$ multiple times, and gets the output.
3. Adversary outputs $b' \in \{0,1\}$, if $b' = b$, adversary wins.

Algorithm 7. Oracles for refresh sub Games

```
    Procedure INIT()
 1: $b \leftarrow_\$ \{0,1\}$
 2: $E \leftarrow_\$ \Pi_{mw}(2n, 2n, \mathcal{K}, \mathcal{W}, \mathsf{sum}, \mathcal{W}_{\mathsf{sum}})$
 3: $k \leftarrow_\$ \{0,1\}^n$ //No usage in M-EXT game

    Procedure M-EXT$[E](inc, h_1, I, I_{cpu})$
 1: if $b = 0$ then
 2:     $s \leftarrow_\$ \{0,1\}^{2n}$
 3: else
 4:     $k \leftarrow \mathrm{COMP}[E](inc, h_1, I)$
 5:     $I_{cpu} \leftarrow 0^n \parallel k \parallel I_{cpu} \parallel 0^{c-1}$
 6:     $s_L \leftarrow \mathrm{COMP}[E](0, const, I_{cpu} \parallel 0)$
 7:     $s_R \leftarrow \mathrm{COMP}[E](0, const, I_{cpu} \parallel 1)$
 8:     $s \leftarrow s_L \parallel s_R$
 9: return $(inc, h_1, s)$

    Procedure pREF$_\mathrm{a}[E](I)$
 1: if $b = 0$ then
 2:     $h \leftarrow_\$ \{0,1\}^n$
 3: else
 4:     $I_0 \leftarrow 0^n \parallel k$
 5:     $(I_1, \ldots, I_\ell) \xleftarrow{2n} I$
 6:     $h \leftarrow const$
 7:     for $i \leftarrow 0$ to $\ell$ do
 8:         $h \leftarrow B[E](h, (i+1) \cdot 2n, I_i)$
 9: return $h$

    Procedure pREF$_\mathrm{f}[E](I, I_{cpu})$
 1: if $b = 0$ then
 2:     $s \leftarrow_\$ \{0,1\}^{2n}$
 3: else
 4:     $y \leftarrow k$
 5:     $y \leftarrow \mathrm{COMP}[E](0, const, 0^n \parallel y \parallel I)$
 6:     $I_{cpu} \leftarrow 0^n \parallel y \parallel I_{cpu} \parallel 0^{c-1}$
 7:     $s_L \leftarrow \mathrm{COMP}[E](0, const, I_{cpu} \parallel 0)$
 8:     $s_R \leftarrow \mathrm{COMP}[E](0, const, I_{cpu} \parallel 1)$
 9:     $s \leftarrow s_L \parallel s_R$
10: return $s$
```

pREF$_\mathrm{a}$ **game.**

1. Oracle runs INIT() procedure.
2. Adversary $\mathcal{A}_2$ queries E, E^{-1} multiple time and $\mathcal{A}_1$ queries pREF$_\mathrm{a}[E](I)$ once, and get returned value. Note that the order of queries is not specified.
3. Adversary outputs $b' \in \{0,1\}$, if $b' = b$, adversary wins.

$\mathrm{pREF_f}$ **game.**

1. Oracle runs INIT() procedure.
2. Adversary $\mathcal{A}_2$ queries E, E^{-1} multiple time and $\mathcal{A}_1$ queries $\mathrm{pREF_f}[E](I, I_{\ cpu})$ once, and get returned value. Note that the order of queries is not specified.
3. Adversary outputs $b' \in \{0,1\}$, if $b' = b$, adversary wins.

And define some values like below.

- $\mathbf{Adv}_{\text{M-EXT}}(p, q, \sigma, \lambda)$: The advantage upper bound against any λ-legitimate adversary $\mathcal{A}$ that makes at most p queries to E or E^{-1}, q queries to M-EXT, and the total length of entropy input I is less than $2n\sigma$ bits.
- $\mathbf{Adv}_{\mathrm{pREF_a}}(p, \ell)$: The advantage upper bound against any adversary $\mathcal{A}$ that makes at most p queries to E or E^{-1}, entropy input I's length for $\mathrm{pREF_a}$ is less than $2n\ell$ bits.
- $\mathbf{Adv}_{\mathrm{pREF_f}}(p, \ell)$: The advantage upper bound against any adversary $\mathcal{A}$ that makes at most p queries to E or E^{-1}, r entropy input blocks to $\mathrm{pREF_f}$, and the input block I and $I_{\ cpu}$'s length is less than $2n\ell$ bits.

Algorithm 8. Oracles for Base ROR subgames

```
    Procedure INIT()
 1: $\mathsf{k_{base}} \leftarrow_\$ \{0,1\}^n$
 2: $b \leftarrow_\$ \{0,1\}$
 3: $\pi \leftarrow_\$ \mathsf{Perm}(2n)$

    Procedure $\mathrm{bROR_k}[\pi](len)$
 1: if $b = 0$ then
 2:     $y_0 \leftarrow_\$ \{0,1\}^{len+2n}$
 3:     return $y_0$
 4: else
 5:     $\mathsf{k_{base}} \| \mathsf{c\_key} \leftarrow \mathsf{CB}[\pi](\mathsf{k_{base}}, 0, 2n)$
 6:     $k \leftarrow \mathsf{c\_key}$
 7:     $\mathsf{c\_key} \| y_1 \leftarrow \mathsf{CB}[\pi](\mathsf{c\_key}, 0, 2n)$
 8:     $B \leftarrow \mathsf{CB}[\pi](k, 1, len - n)$
 9:     $y_1 \leftarrow y_1 \| B$
10:     $y_1 \leftarrow y_1[0 : len - 1]$
11:     return $\mathsf{k_{base}} \| \mathsf{c\_key} \| y_1$

    Procedure $\mathrm{bROR_u}[\pi](len)$
 1: if $b = 0$ then
 2:     $y_0 \leftarrow_\$ \{0,1\}^{len+2n}$
 3:     return $y_0$
 4: else
 5:     $\mathsf{k_{base}} \| \mathsf{c\_key} \leftarrow \mathsf{CB}[\pi](\mathsf{k_{base}}, 0, 2n)$
 6:     $\mathsf{c\_key} \| k \leftarrow \mathsf{CB}[\pi](\mathsf{c\_key}, 0, 2n)$
 7:     if $len \le n$ then
 8:         return $\mathsf{k_{base}} \| \mathsf{c\_key} \| k[0 : len - 1]$
 9:     $y_1 \leftarrow \mathsf{CB}[\pi](k, 1, len)$
10:     $y_1 \leftarrow y_1[0 : len - 1]$
11:     return $\mathsf{k_{base}} \| \mathsf{c\_key} \| y_1$
```

With oracles in Algorithm 8 and in Algorithm 9, we can define $\mathrm{bROR_k}$ game, $\mathrm{bROR_u}$ and $\mathrm{cROR_k}$ game, $\mathrm{cROR_u}$ game processes like below.

Algorithm 9. Oracles for CPU ROR subgames

```
Procedure INIT()
1: $\mathsf{c\_key} \leftarrow_\$ \{0,1\}^n$
2: $b \leftarrow_\$ \{0,1\}$
3: $\pi \leftarrow_\$ \mathsf{Perm}(2n)$
```

```
Procedure $\mathrm{cROR_k}[\pi](len)$
1: if $b = 0$ then
2:     $y_0 \leftarrow_\$ \{0,1\}^{len+2n}$
3:     return $y_0$
4: else
5:     $k \leftarrow \mathsf{c\_key}$
6:     $\mathsf{c\_key} \parallel y_1 \leftarrow \mathsf{CB}[\pi](\mathsf{c\_key}, 0, 2n)$
7:     $B \leftarrow \mathsf{CB}[\pi](k, 1, len - n)$
8:     $y_1 \leftarrow y_1 \parallel B$
9:     $y_1 \leftarrow y_1[0 : len - 1]$
10:    return $\mathsf{k_{base}} \parallel \mathsf{c\_key} \parallel y_1$
```

```
Procedure $\mathrm{cROR_u}[\pi](len)$
1: if $b = 0$ then
2:     $y_0 \leftarrow_\$ \{0,1\}^{len+2n}$
3:     return $y_0$
4: else
5:     $\mathsf{c\_key} \parallel k \leftarrow \mathsf{CB}[\pi](\mathsf{c\_key}, 0, 2n)$
6:     if $len \leq n$ then
7:         return $\mathsf{k_{base}} \parallel \mathsf{c\_key} \parallel k[0 : len - 1]$
8:     $y_1 \leftarrow \mathsf{CB}[\pi](k, 1, len)$
9:     $y_1 \leftarrow y_1[0 : len - 1]$
10:    return $\mathsf{k_{base}} \parallel \mathsf{c\_key} \parallel y_1$
```

$\mathrm{bROR_x}$ **game.**($x \in \{k, u\}$)

1. Oracle runs INIT() procedure.
2. Adversary $\mathcal{A}_2$ queries π, π^{-1} multiple time and queries $\mathrm{bROR_x}[\pi](len)$ once, and get returned value. Note that the order of queries is not specified.
3. Adversary outputs $b' \in \{0,1\}$, if $b' = b$, adversary wins.

$\mathrm{cROR_x}$ **game.**($x \in \{k, u\}$)

1. Oracle runs INIT() procedure.
2. Adversary $\mathcal{A}_2$ queries π, π^{-1} multiple time and queries $\mathrm{cROR_x}[\pi](len)$ once, and get returned value. Note that the order of queries is not specified.
3. Adversary outputs $b' \in \{0,1\}$, if $b' = b$, adversary wins.

And for all $\mathcal{O} \in \{\mathrm{bROR_k}, \mathrm{bROR_u}, \mathrm{cROR_k}, \mathrm{cROR_u}\}$, let $\mathbf{Adv}_{\mathcal{O}}(p, \ell)$ be the advantage upper bound against any adversary $\mathcal{A}$ that makes at most p queries to π or π^{-1}, inputs $2n\ell$ to $\mathcal{O}$.

After hopping every game, we obtain the upper bound of $\Delta_{\mathcal{A}}(S_h, S_1)$. The result is presented in the Lemma 2.

Lemma 2. *For any λ-legitimate robustness adversary $\mathcal{A}$ satisfying conditions in Theorem 1, the following holds.*

$$\begin{aligned}
&\Delta_{\mathcal{A}}(S_0, S_1) \leq 2\Delta_{\mathcal{A}}(S_h, S_1) \\
&\leq 2\mathbf{Adv}_{\text{M-EXT}}(p + 3q_1 + \sigma_1, q_1, \sigma_1, \lambda) \\
&+ 2q_1 \left(\mathbf{Adv}_{\text{pREF}_\text{a}}(p + 3q_1 + \sigma_1, \ell_1) + \mathbf{Adv}_{\text{pREF}_\text{f}}(p + 3q_1 + \sigma_1, \ell_1)\right) \\
&+ 2q_2 \left(\mathbf{Adv}_{\text{bROR}_\text{k}}(p + 2q_2 + \sigma_2, \ell_2) + \mathbf{Adv}_{\text{cROR}_\text{k}}(p + 2q_2 + \sigma_2, \ell_2)\right) \\
&+ 2q_2 \left(\mathbf{Adv}_{\text{bROR}_\text{u}}(p + 2q_2 + \sigma_2, \ell_2) + \mathbf{Adv}_{\text{cROR}_\text{u}}(p + 2q_2 + \sigma_2, \ell_2)\right).
\end{aligned}$$

Here we present a brief overview of the proof. See the full version of this paper [8] for the detailed proof.

To upper bound $\Delta_{\mathcal{A}}(S_1, S_h)$, we introduce intermediate worlds. The bounds in Lemma 2 can be obtained by bounding the advantage between intermediate worlds with subgame advantages. The intermediate worlds are as follows for $1 \leq i \leq q_1$, $1 \leq j \leq q_2$.

- R: For the all $\text{REF}_\text{f}(I, \gamma, I_{cpu})$ immediately after entropy drain, execute $\text{REF}^*_\text{f}(I, \gamma, I_{cpu})$ instead. All other operations are the same as in S_1. $\Delta_{\mathcal{A}}(S_1, R)$ can be bounded by $\mathbf{Adv}_{\text{M-EXT}}(p + 3q_1 + \sigma_1, r, \sigma_1, \lambda)$.
- T_i: For the ith $\text{REF}_\text{f}(I, \gamma, I_{cpu})$, execute $\text{REF}^*_\text{f}(I, \gamma, I_{cpu})$ instead. All other operations are the same as in T_{i-1}, where $T_0 = R$. $\Delta_{\mathcal{A}}(T_{i-1}, T_i)$ can be bounded by $\mathbf{Adv}_{\text{pREF}_\text{f}}(p + 3q_1 + \sigma_1, \ell_1)$.
- W_i: For the ith $\text{REF}_\text{a}(I, \gamma)$, execute $\text{REF}^*_\text{a}(I, \gamma)$ instead. All other operations are the same as in W_{i-1}, where $W_0 = T_r$. $\Delta_{\mathcal{A}}(W_{i-1}, W_i)$ can be bounded by $\mathbf{Adv}_{\text{pREF}_\text{a}}(p + 3q_1 + \sigma_1, \ell_1)$.
- H_i^{Bn}: If the ith ROR is $\text{ROR}_\text{k}[\pi](len, n_{cpu})$ and $S.\mathsf{G\_flag}_{n_{cpu}} = 1$, then execute $\text{ROR}^*_\text{n}[\pi](len, n_{cpu})$ instead. All other operations are the same as in H_{i-1}^C. $\Delta_{\mathcal{A}}(H_{i-1}^C, H_i^{Bn})$ can be bounded by $\mathbf{Adv}_{\text{bROR}_\text{k}}(p + 2q_2 + \sigma_2, \ell_2)$.
- H_i^B: If the ith ROR call is $\text{ROR}_\text{u}[\pi](len, n_{cpu})$ and $S.\mathsf{G\_flag}_{n_{cpu}} = 1$, then execute $\text{ROR}^*_\text{u}[\pi](len, n_{cpu})$ instead. All other operations are the same as in H_i^{Bn}. $\Delta_{\mathcal{A}}(H_i^{Bn}, H_i^B)$ can be bounded by $\mathbf{Adv}_{\text{bROR}_\text{u}}(p + 2q_2 + \sigma_2, \ell_2)$.
- H_i^{Cn}: If the ith ROR is $\text{ROR}_\text{k}[\pi](len, n_{cpu})$ and $S.\mathsf{G\_flag}_{n_{cpu}} = 0$, then execute $\text{ROR}^*_\text{n}[\pi](len, n_{cpu})$ instead. All other operations are the same as in H_i^B. $\Delta_{\mathcal{A}}(H_i^B, H_i^{Cn})$ can be bounded by $\mathbf{Adv}_{\text{cROR}_\text{k}}(p + 2q_2 + \sigma_2, \ell_2)$.
- H_i^C: If the ith ROR is $\text{ROR}_\text{u}[\pi](len, n_{cpu})$ and $S.\mathsf{G\_flag}_{n_{cpu}} = 0$, then execute $\text{ROR}^*_\text{u}[\pi](len, n_{cpu})$ instead. All other operations are the same as in H_i^{Cn}, where $H_0^C = W_r$. $\Delta_{\mathcal{A}}(H_i^{Cn}, H_i^C)$ can be bounded by $\mathbf{Adv}_{\text{cROR}_\text{u}}(p + 2q_2 + \sigma_2, \ell_2)$.

With Lemma 2 and Lemma 3, we can prove the Theorem 1. The proof of the following lemma is in the full version [8].

Lemma 3. *If $p+\sigma \le 2^{n-1}$ and $p+\ell \le 2^{n-1}$, the following inequalities hold:*

$$\mathbf{Adv}_{\text{M-EXT}}(p,q,\sigma,\lambda) \le \frac{9pq+4q\sigma+\sigma^2}{2^n} + \frac{9q}{2^{0.5n}} + \frac{p^2}{2^n} + \frac{4pq}{2^\lambda}, \tag{2}$$

$$\mathbf{Adv}_{\text{pREF}_\text{a}}(p,\ell) \le \frac{3p}{2^n} + \frac{\ell}{2^n} + \frac{3}{2^{0.5n}}, \tag{3}$$

$$\mathbf{Adv}_{\text{pREF}_\text{f}}(p,\ell) \le \frac{9p}{2^n} + \frac{\ell+2}{2^n} + \frac{9}{2^{0.5n}}, \tag{4}$$

$$\mathbf{Adv}_{\text{bROR}_\text{k}}(p,\ell) \le \frac{1+2p}{2^n} + \frac{\ell^2+\ell p}{2^{2n}}, \tag{5}$$

$$\mathbf{Adv}_{\text{cROR}_\text{k}}(p,\ell) \le \frac{p}{2^n} + \frac{\ell^2+\ell p}{2^{2n}}, \tag{6}$$

$$\mathbf{Adv}_{\text{bROR}_\text{u}}(p,\ell) \le \frac{1+3p}{2^n} + \frac{\ell^2+\ell p}{2^{2n}}, \tag{7}$$

$$\mathbf{Adv}_{\text{cROR}_\text{u}}(p,\ell) \le \frac{2p}{2^n} + \frac{\ell^2+\ell p}{2^{2n}}. \tag{8}$$

6 Tight Attacks on Linux-DRBG

In this section, we briefly explain attacks to demonstrate the tightness of our proof. We will present two attacks: the first attack can be executed when $\lambda < n$ with $O(2^{\lambda/2})$ complexity, and the second attack has $O(2^{n/2})$ complexity.

Attack 1: When $\lambda < n$, a λ-legitimate adversary $\mathcal{A}$ can win the robustness game with high probability with the following method.

1. Make $\mathcal{A}_1$ to pick entropy inputs uniformly random from set $\mathcal{T}$ where $|\mathcal{T}| = 2^\lambda$, regardless of query result. Note that $\mathcal{A}$ is still λ-legitimate.
2. For any $S^* \in \mathcal{S}$, and distinct $I_1 \cdots, I_p \in \mathcal{T}$, $\mathcal{A}_2$ simulates $S_i \leftarrow \mathsf{refresh}_\mathsf{f}[E](S^*, I_i, 0^n)$ by repeatedly querying E and calculate p S_i values.
3. For any positive integer c, $\mathcal{A}_2$ simulates $\mathsf{next}_\mathsf{k}[\pi](S_i, 3n, 1)$ by repeatedly querying π and calculates p output random bits. Then save the outputs in $\mathcal{X}$.
4. $\mathcal{A}_2$ queries $\text{SET}(S^*)$ and $\mathcal{A}_1$ picks I and queries $\text{REF}_\text{f}[E](I, \lambda, 0^n)$, then $\mathcal{A}_2$ makes $\text{ROR}_\text{u}[\pi](3n, 1)$ to get random bits. Repeat this procedure q times and save the values in $\mathcal{Y}$.
5. If $\mathcal{X} \cap \mathcal{Y} = \emptyset$, $\mathcal{A}_2$ outputs 0. Else, $\mathcal{A}_2$ outputs 1.

To make an intersection, in the real world, it is sufficient to make the collision between entropy input and simulated entropy input. However, in the ideal world, the output bits are generated uniformly randomly. Therefore we have

$$\Pr[1 \leftarrow \mathcal{A} \mid b = 0] = \frac{pq}{2^{3n}}$$

$$\Pr[1 \leftarrow \mathcal{A} \mid b = 1] = 1 - \left(1 - \frac{p}{2^\lambda}\right)^q \ge \frac{pq}{2^\lambda} - \frac{(pq)^2}{2^{2\lambda+1}}.$$

Therefore, if $p = q = 2^{\lambda/2}$, the advantage of $\mathcal{A}$ is sufficiently non-negligible.

Attack 2: A λ-legitimate adversary $\mathcal{A}$ can win a robustness game with high probability with the following method.

1. $\mathcal{A}$ picks key $K \leftarrow_{\$} \{0,1\}^n$ and $\mathcal{A}$ simulates $\mathsf{next_k}[\pi](S, 3n, 1)$ as if $S.\mathsf{ckey}_1 = K$ and $S.\mathsf{G\_flag}_1 = 0$ by repeatedly querying π. Then save the outputs in $\mathcal{X}$. Repeat this procedure p times.
2. $\mathcal{A}_1$ generates I with min-entropy λ, then queries $\mathrm{REF}_\mathsf{f}[E](I, \lambda, 0^n)$ and $\mathcal{A}_2$ queries $\mathrm{ROR}_\mathsf{k}[\pi](3n, 1)$ and save the outputs in $\mathcal{Y}$. Repeat this procedure until $|\mathcal{Y}|$ becomes q.
3. If $\mathcal{X} \cap \mathcal{Y} = \emptyset$, $\mathcal{A}$ outputs 0. Else, $\mathcal{A}$ outputs 1.

To make an intersection, in the real world, it is sufficient to make the collision on $S.\mathsf{ckey}_1$. However, in the ideal world, the output bits are generated uniformly randomly. Therefore we have

$$\Pr[1 \leftarrow \mathcal{A} \mid b = 0] = \frac{pq}{2^{3n}}$$
$$\Pr[1 \leftarrow \mathcal{A} \mid b = 1] = 1 - \left(1 - \frac{q}{2^n}\right)^p \geq \frac{pq}{2^n} - \frac{(pq)^2}{2^{2n+1}}.$$

Therefore, if $p = q = 2^{n/2}$, the advantage of $\mathcal{A}$ is sufficiently non-negligible.

Acknowledgements. Jooyoung Lee was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) [NO.2022-0-01047, Development of statistical analysis algorithm and module using homomorphic encryption based on real number operation]. We would like to thank Samuel Neves for informing us of the recent research on weak states in BLAKE2, which contributed to our modeling and security proof.

References

1. J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. Blake2: simpler, smaller, fast as md5. In *Applied Cryptography and Network Security: 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings 11*, pages 119–135. Springer, 2013.
2. B. Barak and S. Halevi. A Model and Architecture for Pseudo-Random Generation with Applications to /Dev/Random. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 203–212, New York, NY, USA, 2005. Association for Computing Machinery.
3. D. J. Bernstein et al. Chacha, a variant of salsa20.
4. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge-based pseudo-random number generators. In *Cryptographic Hardware and Embedded Systems, CHES 2010: 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings 12*, pages 33–47. Springer, 2010.
5. A. Biryukov, A. Udovenko, and V. Velichkov. Analysis of the norx core permutation. *Cryptology ePrint Archive*, 2017.

6. M. J. Campagna. Security bounds for the nist codebook-based deterministic random bit generator. Cryptology ePrint Archive, Paper 2006/379, 2006. https://eprint.iacr.org/2006/379.
7. C. Chaigneau, T. Fuhr, H. Gilbert, J. Jean, and J.-R. Reinhard. Cryptanalysis of norx v2. 0. *Journal of Cryptology*, 32:1423–1447, 2019.
8. W. Chung, H. Kim, J. Lee, and Y. Lee. Provable security of Linux-DRBG in the seedless robustness model. Cryptology ePrint Archive, Paper 2024/1421, 2024.
9. S. Coretti, Y. Dodis, H. Karthikeyan, N. Stephens-Davidowitz, and S. Tessaro. On seedless prngs and premature next. *Cryptology ePrint Archive*, 2022.
10. S. Coretti, Y. Dodis, H. Karthikeyan, and S. Tessaro. Seedless fruit is the sweetest: Random number generation, revisited. In *Annual International Cryptology Conference*, pages 205–234. Springer, 2019.
11. J. P. Degabriele, J. Govinden, F. Günther, and K. G. Paterson. The security of chacha20-poly1305 in the multi-user setting. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1981–2003, New York, NY, USA, 2021. Association for Computing Machinery.
12. Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, and D. Wichs. Security Analysis of Pseudo-Random Number Generators with Input: /Dev/Random is Not Robust. CCS '13, page 647–658, New York, NY, USA, 2013. Association for Computing Machinery.
13. P. Gaži and S. Tessaro. Provably robust sponge-based prngs and kdfs. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I 35*, pages 87–116. Springer, 2016.
14. F. Goichon, C. Lauradoux, G. Salagnac, and T. Vuillemin. *Entropy transfers in the Linux random number generator*. PhD thesis, INRIA, 2012.
15. Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the linux random number generator. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–pp. IEEE, 2006.
16. S. Hirose. Security analysis of drbg using hmac in nist sp 800-90. In K.-I. Chung, K. Sohn, and M. Yung, editors, *Information Security Applications*, pages 278–291, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
17. V. T. Hoang and Y. Shen. Security analysis of nist ctr-drbg. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part I*, pages 218–247. Springer, 2020.
18. A. Luykx, B. Mennink, and S. Neves. Security analysis of blake2's modes of operation. *IACR Transactions on Symmetric Cryptology*, pages 158–176, 2016.
19. S. Müller. *Documentation and analysis of the linux random number generator*. Federal Office for Information Security, 2020.
20. S. Ruhault. Sok: Security models for pseudo-random number generators. *IACR Transactions on Symmetric Cryptology*, pages 506–544, 2017.
21. T. Shrimpton and R. S. Terashima. A provable-security analysis of intel's secure key rng. In *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 77–100. Springer, 2015.
22. T. Shrimpton and R. S. Terashima. Salvaging weak security bounds for blockcipher-based constructions. In *ASIACRYPT (1)*, pages 429–454. Springer, 2016.
23. J. Woodage and D. Shumow. An analysis of nist sp 800-90a. 11477:151–180, 2019.

24. J. Woodage and D. Shumow. An analysis of nist sp 800-90a. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part II 38*, pages 151–180. Springer, 2019.
25. K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel. Verified correctness and security of mbedtls hmac-drbg. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2007–2020, 2017.

Author Index

K.-M. Chung and Y. Sasaki (Eds.): ASIACRYPT 2024, LNCS 15489, pp. 491–492, 2025.
https://doi.org/10.1007/978-981-96-0938-3