MPC FOR EVERYONE

by

Aarushi Goel

A dissertation submitted to The Johns Hopkins University in conformity
with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland
July, 2022

© 2022 Aarushi Goel
All rights reserved

Abstract

The age of internet of things, where each device and application double up as a source of data has led to an unprecedented influx of data and analyzing this data is becoming increasingly useful. Given its sensitive nature, there is a growing demand for better and more efficient data collection and computation techniques that respect privacy. Most existing techniques for privacy-preserving computations incur large overheads, limiting platforms that can be used for performing such heavy-duty computations. Moreover, using such platforms for all computations, accumulates power with organizations that own these platforms and creates central targets of failure. This necessitates the need for distributing work and power when computing on private data.

Powerful and well-studied cryptographic notions such as secure multiparty computation (MPC) help distribute power by enabling privacy-preserving collaboration between mutually distrusting entities for complex computations on data. Unfortunately, modern MPC protocols have unaccommodating participation models. In general, parties participating in such protocols are required to perform large computations and are expected to stay active throughout the execution. However, unlike large organizations, not everyone might have the resources to carry out such large-scale and long-drawn computations. In this dissertation, our goal is to democratize such computations by designing MPC protocols that empower regular people and smaller organizations to emulate large-scale computations in a distributed manner. We make progress in two different directions.

In the first part of this dissertation, we incentivize more participation in an MPC protocol by effectively "distributing" the work amongst parties. In most known protocols, computation and communication amongst parties increases as the number of participants increase. We propose a new

MPC protocol, where the per-party work decreases as the number of parties increase. As a result, when run with a large number of parties, the burden on each individual participant is significantly reduced – enabling efficient large-scale MPC computations, involving hundreds and thousands of participants. Including more participants also dilutes the power of each individual party, which is highly desirable.

In the second part of this dissertation, we introduce a new participation model called Fluid MPC. Unlike all existing protocols, where participants are required to remain online throughout the execution, in this model, one can design protocols that allow parties to leave and join the protocol execution as they wish. The minimum amount of work that a party is required to do in order to participate is extremely small in comparison to the size of the entire computation. This extreme flexibility allows parties – including those with low resources and limited time – to contribute according to their computational capacity and effectively yields a weighted, privacy-preserving, distributed computing system.

Thesis Readers

Dr. Abhishek Jain (Primary Advisor)
Associate Professor, Dept. of Computer Science, Johns Hopkins University

Dr. Matthew Green
Associate Professor, Dept. of Computer Science, Johns Hopkins University

Dr. Sanjam Garg
Associate Professor, Dept. of Electrical Engineering and Computer Sciences, UC, Berkeley
Senior Scientist, NTT Research

Dedicated to my incredible father and to the loving memory of my late mother.

Acknowledgments

I have had the most wonderful and memorable experience as a grad student at Hopkins, thanks to *several* amazing people.

Firstly, I would like to thank my advisor Abhishek Jain. Abhishek is an incredible mentor and I was extremely fortunate to be his student. I am grateful to him for being so supportive and patient with me and for constantly motivating me to be more confident in my own research skills. He has taught me how to extract and articulate logical ideas from seemingly incoherent and abstract thoughts and how to approach research more systematically. Abhishek is always open to new ideas and the range of topics (and problems) that he is working on at any given time is truly inspiring. I can't thank him enough for all the opportunities and his invaluable advice on life and career. I really enjoyed being his student and I hope to get more opportunities to continue learning from him in the future.

Next, I would like to thank Matthew Green. Matt's ability to always remain up-to-date with everything security-related, happening around the world is remarkable. I have greatly benefitted the numerous open-ended questions that he would often ask during lab meetings and on Slack, that inevtiably led to long group discussions. I was lucky to be able to work with him on several projects and learn from his impressive breadth of knowledge on both applied and theoretical cryptography. Both Matt and Abhishek really lookout for all the students at Hopkins and I thank both of them for providing all of us with a positive and collaborative research lab enviroment.

During my PhD, I had the opportunity to have been hosted for two research visits to Israel by Elette Boyle at IDC, Herzliya and Benny Applebaum at Tel Aviv University respectively. Elette is a

brilliant researcher and the most fun person I have worked with. It was amazing to see how despite her busy schedule, she somehow managed to find time to meet with all the interns almost everyday. Outside of work, Elette went out of her way to make sure I felt welcomed in Israel and had the most amazing summers during both my visits. I also want to thank everyone else who I met at IDC, especially Alon Rosen, Tal Moran, Lisa Kohl and Panos Charalampopoulos who were incredibly kind and fun to be around. Benny is one of the smartest people I have met. He has contributed greatly in my understanding of fundamental concepts and has taught me how to write better proofs. I am sincerely thankful to both Elette and Benny for giving me the oppotunity to work with them and for making my visits productive and worth-while.

I was fortunate to spend my final year at UC Berkeley and I am grateful to Sanjam Garg for enabling this year-long visit. Sanjam is a constant source of inpiration and I have learnt a great deal from him in this past year. I thank him for being immensely supportive and kind to me. I look forward to working with and learning more from him during my postdoc. I would also like to thank all the other students, postdocs and visitors who made my time at Berkeley extremely memorable – Guru Vamsi Policharla, Yinou Zhang, Jaiden Fairoze, James Bartusek, Bhaskar Roberts, Sruthi Sekar, Mingyuan Wang, Sina Shiehian, Pedro Branco, Doreen Riepel, Arka Rai Choudhuri and Zhengzhong Jin. I must give a special shout-out to my officemates at Berkeley – Sruthi and Mingyuan – for being such great company and for making me look forward to coming to office everyday. The three of us have spent countless hours chatting away in the office about anything and everything.

I would like to thank all of my collaborators for all the work that they put into the projects – Prabhanjan Ananth, Benny Applebaum, Gabrielle Beck, Elette Boyle, Arka Rai Choudhuri, Ran Cohen, Harry Eldridge, Sanjam Garg, Matthew Green, Mathias Hall-Andersen, Aditya Hegde, Abhishek Jain, Zhengzhong Jin, Gabriel Kaptchuk, Manoj Prabhakaran, Rajeev Raghunath, Nicholas Spooner and Maximilian Zinkus. I must especially thank Prabhanjan, with whom I worked with at the beginning of my PhD, for patiently answering all my stupid questions and Ran, for teaching me how to write better papers and Latex code.

My experience in grad school was especially made memorable by all of my labmates at Hopkins

– Gabby, Alishah, Arka, Harry, Nils, Christina, Aditya, Zhengzhong, Tushar, Gabriel, Ian, Pratyush, Gijs and Max. I thank them for being the best colleagues one could ask for. A special thanks goes to Arka, Alishah, Gabe and Gabby. Arka, and Alishah started their PhDs at the same time as me and they have been great friends ever since. Over the years, I have learnt so much from them and I cherish all our technical and (mostly) non-technical conversations about everything ranging from course projects, research papers, to lessons on pop-culture and their never-ending list of movie recommendations. Gabby is one of the nicest people I have come across and a very dear friend. I hope we continue to stay in touch via our (now not so regular!) weekend phone calls. Gabe has been the most amazing collaborator and an even better friend. Our prolonged discussions, multiple days a week, have played an important role in my evolution as a researcher. I thank him for always patiently lending an ear to my endless rants.

Outside of the cryptography group, I was lucky to have also made friends with other students in the computer science department – Yasamin Nazari, Aditya Krishnan, Ama Koranteng, Ravi Shankar and Enayat Ullah. They have all been a great support system at different times during my PhD. Especially during COVID, I cannot imagine surviving first year or so of lockdowns without their support.

Finally, I want to thank my friends and family to their continued and unconditional support. My friends – Nikita, Jasmine, Alankrita, Akshima and Meenakshi – who I have known for several years now and who have helped me stay sane throughout my PhD. My late mother, who is my biggest role model and whose memories are a constant source of encouragement in my life. Lastly, I would like to thank my incredible father and baby brother for supporting me in all my decisions and for their unconditional love.

Contents

Abstract ii

Acknowledgments v

List of Tables xi

List of Figures xii

1 Introduction 1
- 1.1 Order-C Secure Multiparty Computation . . . 3
  - 1.1.1 Our Contributions . . . 6
- 1.2 Secure Multiparty Computation with Dynamic Participants . . . 9
  - 1.2.1 Our Contributions . . . 13
  - 1.2.2 Related Work . . . 14
- 1.3 Bibliographic Notes . . . 16
- 1.4 Outline of the Thesis . . . 16

2 Preliminaries 18
- 2.1 Secure Multiparty Computation . . . 18
  - 2.1.1 Adversarial Behavior . . . 18
  - 2.1.2 Security Definitions . . . 19
- 2.2 Secret Sharing . . . 21
  - 2.2.1 Threshold Secret Sharing . . . 21
  - 2.2.2 Packed Secret Sharing . . . 22

3 Order-C Secure Mutliparty Computation 24
- 3.1 Technical Overview . . . 24
  - 3.1.1 Background . . . 24
  - 3.1.2 Our Approach: Semi-Honest Security . . . 25
  - 3.1.3 Malicious Security . . . 32
- 3.2 Preliminaries . . . 33
- 3.3 Highly Repetitive Circuits . . . 35
  - 3.3.1 Wire Configuration . . . 35
  - 3.3.2 (A, B)-Repetitive Circuits . . . 36
  - 3.3.3 Examples of Highly Repetitive Circuits . . . 38

3.3.4 Protocol Switching for Circuits with Partially Repeated Structure 42
3.4 Input Sharing Phase 43
3.4.1 Generating Shares of Random Values 43
3.4.2 Secret Sharing of Inputs 45
3.4.3 A Non-Interactive Protocol for Packing Regular Secret Shares 47
3.4.4 Packed Secret Sharing of Inputs 49
3.5 Circuit Evaluation Phase 50
3.5.1 Generating Correlated Random Packed Sharings 51
3.5.2 Secure Layer-Wise Circuit Evaluation 53
3.6 Our Order-C Semi-Honest Protocol 55
3.7 Our Order-C Maliciously Secure Protocol 58
3.7.1 Generating Random Packed Shares 58
3.7.2 Checking Equality to Zero 60
3.7.3 Secure Dual Evaluation upto Linear Attacks 61
3.7.4 Secure Multiplication upto Linear Attacks 63
3.7.5 Maliciously Secure Protocol 64
3.8 Security Proof for our Maliciously Secure Protocol 66
3.9 Implementation and Evaluation 75
3.9.1 Comparison 75
3.9.2 Implementation 76

4 Secure Multiparty Computation with Dynamic Participants **80**
4.1 Technical Overview 80
4.1.1 Main Challenges 81
4.1.2 Adapting Optimized Semi-honest BGW [GRR98] to Fluid MPC 82
4.1.3 Compiler for Malicious Security 84
4.2 Fluid MPC 90
4.2.1 Modeling Dynamic Computation 91
4.2.2 Committees 92
4.2.3 Security 96
4.3 Preliminaries 105
4.3.1 Layered Circuits 105
4.4 Roadmap to Our Results 107
4.5 Additive Attack Paradigm in Fluid MPC 108
4.5.1 Linear-Based Fluid MPC Protocols 109
4.5.2 Weak Privacy and Security up to Additive Attacks 114
4.6 Malicious Security Compiler for Fluid MPC 122
4.6.1 Robust Circuit 123
4.6.2 Maliciously Secure Fluid MPC 125
4.6.2.1 Checking Equality to Zero 126
4.6.2.2 Compiled Protocol 126
4.7 Weakly Private Fluid MPC 136
4.7.1 Linear Protocols 136
4.7.2 Proof of Weak Privacy 138

4.8 Implementation and Evaluation 140
4.8.1 Evaluation 142

Bibliography **144**

List of Tables

3.1 Size of the highly repetitive circuits we consider in this work. 40
3.2 Comparing the runtime of our order-C protocol and that of related work. All times are in milliseconds. 77
4.1 Computation time for Fluid MPC, in milliseconds, per layer of the circuit. 142

List of Figures

1.1 Computation model of fluid MPC. 12

3.1 A simple example pair of circuit layers illustrating the need for differing-operation packed secret sharing and our realignment procedure. 27
3.2 The function $\mathsf{WireConfiguration}(\mathsf{block}_{m+1}, \mathsf{block}_m)$ that computes a proper alignment for computing block_{m+1} 36
3.3 Random share generation functionality 44
3.4 Secret sharing of inputs functionality 46
3.5 Packed Secret sharing of all inputs functionality 49
3.6 A Protocol for Layer-wise Circuit Evaluation 54
3.7 Packed random share generation functionality 59
3.8 Random share generation functionality 60
3.9 Secure Multiplication Up to Linear Attack functionality 63

4.1 **Left:** Part of the circuit partitioned into different layers, indicated by the different colors. **Right:** A visual representation of the flow of information during the modified version of BGW presented in Section 4.1.2. 82
4.2 Epochs ℓ and $\ell + 1$ 92
4.3 Functionality for Committee Formation. 93
4.4 Functionality for checking equality to zero 126
4.5 Fluid Sub-Protocol π_{rand} 137
4.6 Fluid Sub-Protocol π_{mult} 137
4.7 Fluid Sub-Protocol π_{trans} 138
4.8 The computation phase runtimes of circuits with depths 10 (red), 100 (orange) and 1000 (yellow), but approximately equal numbers of multiplication gates. 141

Chapter 1

Introduction

With the advent of big data, there is an unprecedented influx of data. More so now, in the age of internet of things, where each device and application doubles up as a source of data. Analysing this data is becoming increasingly useful in machine learning, healthcare, targeted advertising, etc. As services become more useful, the vast collection of personal data is inevitably *concentrating power* in the hands of the collectors - private firms or the government. Given the sensitive nature of personal data, an inherent trust is placed on these organisations, which is very easily violated. As public concern over data privacy grows, there is an increased demand for better and more efficient data collection and computation techniques that mitigate the trust assumption and reliance on any individual or entity.

The most commonly used cryptographic tool in the data industry is encryption. While sending and storing data in encrypted form certainly ensures privacy, computing on encrypted data is not as easy. In particular, it is quite costly and while these techniques are becoming increasingly more efficient – adding accountability to this computation – still incurs significant overhead. Nevertheless, cloud-computing services such as Google Cloud Platform (GCP) and Amazon Web Services (AWS), that have enabled the internet to flourish are capable of performing heavy-duty computations. However, using such platforms for all computations continues to accumulate power with these large organizations and creates a central point of failure. This necessitates the need for distributing work

and power when computing on private data.

Powerful and well-studied cryptographic notions such as MPC or secure multiparty computation [Yao86, GMW87, CCD88, BGW88] helps distribute power by enabling collaboration between mutually distrusting entities for evaluating complex functions on data while still preserving privacy. Unfortunately, modern MPC protocols have unaccommodating participation models. In general, parties participating in such protocols are required to perform large computations and are expected to stay active throughout the execution. However, unlike large organizations, not everyone might have the resources to carry out such large-scale and long-drawn computations. Our goal is to democratize such computations by designing MPC protocols that empower regular people and smaller organizations to emulate large-scale computations in a distributed manner.

Order-C Total Work. Including more participants in an MPC protocol dilutes the power of each individual party and is highly desirable. Unfortunately, in most known (and all implemented) protocols, the communication and computation complexity increases as the number of participants increase. This inhibits smaller devices (e.g. mobile phones) and participants with fewer computational resources from participating, especially over *low-bandwidth* networks. In the first part of this dissertation, we propose a new MPC protocol, where the per-party communication and computation complexity decreases as the number of parties increase. As a result, when run with a large number of players, the burden on each individual participant is significantly reduced – enabling efficient large-scale MPC computations involving hundreds and thousands of parties. This is the first instance where such a property is achieved for a non-trivial class of functions, including many important applications of MPC such as training algorithms used to create machine learning models. We demonstrate practicality of our approach by implementing this protocol and testing it for hundreds of participants.

Dynamic Participants. In the second part of this dissertation, we introduce a new model of participation called Fluid MPC. As discussed earlier, all existing protocols require participants to remain online throughout the execution, and in case they need to drop out of the protocol

before it completes, the protocol fails. In the Fluid MPC model, one can design protocols that allow parties to leave and join the protocol execution as they wish. The minimum amount of work that a party is required to do in order to participate is extremely small in comparison to the size of the entire computation. This extreme flexibility allows parties – including those with low resources and limited time – to contribute according to their computational capacity and effectively yields a weighted, privacy-preserving, distributed computing system.

We now delve into these problems in more detail.

1.1 Order-C Secure Multiparty Computation

In recent years, MPC techniques are being applied to an increasingly complex class of functionalities such as distributed training of machine learning networks. Most current applications of MPC, however, focus on using a *small* number of parties. This is largely because most known (and all implemented) protocols incur a linear multiplicative overhead in the number of players in the communication and computation complexity, *i.e.* have complexity $O(n|C|)$[1], where n is the number of players and $|C|$ is the size of the circuit [HN06, DN07, LN17, CGH+18, NV18, FL19].

The Need for Large-Scale MPC. Yet, the most exciting MPC applications are at their best when a *large* number of players can participate in the protocol. These include crowd-sourced machine learning and large scale data collection, where widespread participation would result in richer data sets and more robust conclusions. Moreover, when the number of participating players is large, the honest majority assumption – which allows for the most efficient known protocols till date – becomes significantly more believable. Indeed, the honest majority of resources assumptions (a different but closely related set of assumptions) in Bitcoin [Nak08] and TOR [RSG98, DMS04] appear to hold up in practice when there are many protocol participants.

Furthermore, large-scale volunteer networks have recently emerged, like Bitcoin and TOR, that regularly perform incredibly large distributed computations. In the case of cryptocurrencies, it would

[1] For sake of simplicity, throughout the introduction, we omit a linear multiplicative factor of the security parameter in all asymptotic notations.

be beneficial to apply the computational power to more interesting applications than mining, including executions of MPC protocols. Replicating a fraction of the success of these networks could enable massive, crowd-sourced applications that still respect privacy. In fact, attempts to run MPC on such large networks have already started [WJS+19], enabling novel measurements.

Our Goal: Order-C MPC. It would be highly advantageous to go beyond the limitation of current protocols and have access to an MPC protocol with total complexity $O(|C|)$.[2]

Such a protocol can support division of the total computation among players which means that using large numbers of players can significantly reduce the burden on each individual participant. In particular, when considering complex functions, with circuit representations containing tens or hundreds of millions of gates, decreasing the workload of each individual party can have a significant impact. Ideally, it would be possible for the data providers themselves, possibly using low power or bandwidth devices, to participate in the computation.

An $O(|C|)$ MPC protocol can also offer benefits in the design of other cryptographic protocols. In [IKOS07], Ishai et al. showed that zero-knowledge (ZK) proofs [GMR85] can be constructed using an "MPC-in-the-head" approach, where the prover simulates an MPC protocol in their mind and the verifier selects a subset of the players views to check for correctness. The efficiency of these proofs is inherited from the complexity of the underlying MPC protocols, and the soundness error is a function of the number of views opened and the number of players for which a malicious prover must have to "cheat" in order to control the protocol's output. This creates a tension: higher number of players can be used to increase the soundness of the ZK proof, but simulating additional players increases the complexity of the protocol. Access to an $O(|C|)$ MPC protocol would ease this tension, as a large numbers of players could be used to simulate the MPC without incurring additional cost.

Despite numerous motivations and significant effort, there are no known $O(|C|)$ MPC protocols for "non-SIMD" functionalities.[3] We therefore ask the following:

[2] Note that in all existing efficient honest majority protocols that make use of polynomial-based threshold secret sharing, the computation complexity is at least $O(\log n)$ times the communication complexity. This is also true for our protocol. In this introduction, unless stated otherwise, we use expressions "protocol with total complexity $O(X)$" or "$O(|X|)$ protocols" to refer to protocols where the communication complexity is $O(|X|)$ and the computation complexity is $O(X \log n)$. We will discuss the source of this additional term in the computation complexity in more detail later in the main chapters.

[3] SIMD circuits are arithmetic circuits that simultaneously evaluate ℓ copies of the same arithmetic circuit on different

Is it possible to design an MPC protocol with $O(|C|\log n)$ total computation (supporting division of labor) and $O(|C|)$ total communication?

Prior Work: Achieving $\tilde{O}(|C|)$-MPC. A significant amount of effort has been devoted towards reducing the asymptotic complexity of (honest-majority) MPC protocols, since the initial $O(n^2|C|)$ protocols [BGW88, CCD88].

Over the years, two primary techniques have been developed for reducing protocol complexity. The first is an *efficient multiplication protocol* combined with batched correlated randomness generation introduced in [DN07]. Using this multiplication protocol reduces the (amortized) complexity of a multiplication gate from $O(n^2)$ to $O(n)$, effectively shaving a factor of n from the protocol complexity. The second technique is *packed secret sharing* (PSS) [FY92], a vectorized, single-instruction-multiple-data (SIMD) version of traditional threshold secret sharing. By packing $\Theta(n)$ elements into a single vector, $\Theta(n)$ operations can be performed at once, reducing the protocol complexity by a factor of n when the circuit structure is accommodating to SIMD operations. Using these techniques separately, $O(n|C|)$ protocols were constructed in [DI06] and [DN07].

While it might seem as though combining these two techniques would result in an $O(|C|)$ protocol, the structural requirements of SIMD operations make it unclear on how to do so. The works of [DIK+08] and [DIK10] demonstrate two different approaches to combine these techniques, either by relying on randomizing polynomials or using circuit transformations that involve embedding routing networks within the circuits. These approaches yield $\widetilde{O}(|C|)$ protocols with large multiplicative constants and additive terms that depend on the circuit depth. (The additive terms were further reduced in the recent work of [GIP15].)

In summary, while both PSS and efficient multiplication techniques have been known for over a decade, no $O(|C|)$ MPC protocols are known. The best known asymptotic efficiency is $\widetilde{O}(|C|)$ achieved by [DIK+08, DIK10, GIP15]; however, these protocols have never been implemented for reasons discussed above. Instead, the state-of-the-art implemented protocols achieve $O(n|C|)$ computational and communication efficiency [CGH+18, NV18, FL19].

inputs. Genkin et al. [GIP15] showed that it is possible to design an $O(|C|)$ MPC protocol for SIMD circuits, where $\ell = \Theta(n)$.

1.1.1 Our Contributions

In this work, we identify a meaningful class of circuits, called (A, B)-repetitive circuits, parameterized by variables A and B. We show that for $(\Omega(n), \Omega(n))$-repetitive circuits, efficient multiplication and PSS techniques can indeed be combined, using new ideas, to achieve $O(|C|)$ MPC for n parties. To the best of our knowledge, this is the first such construction for a larger class of circuits than SIMD circuits.

We test the practical efficiency of our protocol by means of a preliminary implementation and show via experimental results that for computations involving large number of parties, our protocol outperforms the state-of-the-art implemented MPC protocols. We now discuss our contributions in more detail.

Highly Repetitive Circuits. The class of (A, B)-repetitive circuits are circuits that are composed of an arbitrary number of *blocks* (sets of gates at the same depth) of width at least A, that recur at least B times throughout the circuit. Loosely speaking, we say that an (A, B)-repetitive circuit is *highly repetitive* w.r.t. n parties, if $A \in \Omega(n)$ and $B \in \Omega(n)$.

The most obvious example of this class includes the sequential composition of some (possibly multi-layer) functionality, i.e. $f(f(f(f(\ldots))))$ for some arbitrary f with sufficient width. However, this class also includes many other types of circuits and important functionalities. For example, as we discuss in Section 3.3.3, machine learning model training algorithms (many iterations of gradient descent) are highly repetitive even for large numbers of parties. We also identify block ciphers and collision resistant hash functions as having many iterated rounds; as such functions are likely to be run many times in a large-scale, private computation, they naturally result in highly repetitive circuits for larger numbers of parties. We give formal definition of (A, B)-repetitive circuits in Section 3.3.

Semi-Honest Order-C MPC. Our primary contribution is a semi-honest, honest-majority MPC protocol for highly repetitive circuits with *total complexity* $O(|C|)$. Our protocol only requires communication over point-to-point channels and works in the plain model (i.e., without trusted setup). It

achieves unconditional security against $t < n\left(\frac{1}{2} - \frac{2}{\epsilon}\right)$ corruptions, where ϵ is a tunable parameter as in prior works based on PSS.

Our key insight is that the repetitive nature of the circuit can be leveraged to efficiently generate correlated randomness in a way that helps overcome the limitations of PSS. We elaborate on our techniques in Section 3.1.

Malicious Security Compiler. We next consider the case of malicious adversaries. In recent years, significant work [GIP$^+$14, GIP15, LN17, CGH$^+$18, NV18, FL19, GSZ20] has been done on designing efficient malicious security compilers for honest majority MPC. With the exception of [GIP15], all of these works design compilers for protocols based on regular secret sharing (SS) as opposed to PSS. The most recent of these works [CGH$^+$18, NV18, FL19, GSZ20] achieve very small constant multiplicative overhead, and ideally one would like to achieve similar efficiency in the case of PSS-based protocols.

Since our semi-honest protocol is based on PSS, the compilers of [CGH$^+$18, NV18, FL19, GSZ20] are not directly applicable to our protocols. Nevertheless, borrowing upon the insights from [GIP15], we demonstrate that the techniques developed in [CGH$^+$18] can in fact be used to design an efficient malicious security compiler for our PSS-based semi-honest protocol. Specifically, our compiler incurs a multiplicative overhead of approximately 1.6–2.3, depending on the choice of ϵ, over our semi-honest protocol for circuits over large fields (where the field size is exponential in the security parameter).[4] For circuits over smaller fields, the multiplicative overhead incurred is $O(k/\log|\mathbb{F}|)$, where k is the security parameter and $|\mathbb{F}|$ is the field size.

Efficiency. We demonstrate that our protocol is not merely of theoretical interest but is also concretely efficient for various choices of parameters. We give a detailed complexity calculation of our protocols in Sections 3.6 and 3.7.5.

For $n = 125$ parties and $t < n/3$, our malicious secure protocol only requires each party to, on average, communicate approximately $2\frac{3}{4}$ *field elements per gate* of a highly repetitive circuit.

[4]We note that for more commonly used corruption thresholds $n/2 > t > n/4$, the overhead incurred by our compiler is approximately 2.3.

In contrast, the state-of-the-art [FL19] (an information-theoretic $O(n|C|)$ protocol for $t < n/3$) requires each party to communicate approximately $4\frac{2}{3}$ field elements per multiplication gate. Thus, (in theory) we expect our protocol to outperform [FL19] for circuits with around 65% multiplication gates with just 125 parties. Since the per-party communication in our protocol decreases as the number of parties increase, our protocol is expected to perform better as the number of parties increase.

We confirm our conjecture via a preliminary implementation of our malicious secure protocol and give concrete measurements of running it for up to 300 parties, across multiple network settings. Since state-of-the-art honest-majority MPC protocol have only been tested with smaller numbers of parties, we show that our protocol is comparably efficient even for fewer number of parties. Moreover, our numbers suggest that our protocol would outperform these existing protocols when executed with hundreds or thousands of players at equivalent circuit depths.

Application to Zero-Knowledge Proofs. The *MPC-in-the-head* paradigm of Ishai et al. [IKOS07] is a well-known technique for constructing efficient three-round public-coin honest-verifier zero-knowledge proof systems (aka sigma protocols) from (honest-majority) MPC protocols. Such proof systems can be made *non-interactive*, in the random oracle model [BR93] via the Fiat-Shamir paradigm [FS87]. Recent works have demonstrated the practical viability of this approach by constructing zero-knowledge proofs [GMO16, CDG$^+$17, KKW18, AHIV17] where the proof size has linear or sub-linear dependence on the size of the relation circuit.

Our malicious-secure MPC protocol can be used to instantiate the MPC-in-the-head paradigm when the relation circuit has highly repetitive form. The size of the resulting proofs will be comparable to the best-known *linear-sized* proof system constructed via this approach [KKW18]. Importantly, however, it can have more efficient prover and verifier computation time. This is because [KKW18] requires parallel repetition to get negligible soundness, and have computation time linear in the number of simulated players. Our protocol (by virtue of being an Order-C and honest majority protocol), on the other hand, can accommodate massive numbers of (simulated) parties without

increasing the protocol simulation time and achieve small soundness error without requiring additional parallel repetition. Finally, we note that sublinear-sized proofs [AHIV17] typically require super-linear prover time, in which case simulating our protocol may be more computationally efficient for the prover. We leave further exploration of this direction for future work.

1.2 Secure Multiparty Computation with Dynamic Participants

Given the increasing popularity of MPC, it is inevitable that more ambitious applications will be explored in the near future — like complex simulations on secret initial conditions or training machine learning algorithms on *massive, distributed datasets*. Because the circuit representations of these functionalities can be extremely deep, evaluating them could take several hours or even days, even with highly efficient MPC protocols. While MPC has been studied in a variety of settings over the years, nearly all previous work considers *static* participants who must commit to participating for the entire duration of the computation. However, this requirement may not be reasonable for large, long duration computations such as above because the participants may be limited in their computational resources or in the amount of time that they can devote to the computation at a stretch. Indeed, during such a long period, it is more realistic to expect that some participants may go offline either to perform other duties (or undergo maintenance), or due to connectivity problems.

To accommodate increasingly complex applications and participation from parties with fewer computational resources, MPC protocols must be designed to support *flexibility*. In this work, we formalize the study of MPC protocols that can support *dynamic* participation – where parties can join and leave the computation without interrupting the protocol. Not only would this remove the need for parties to commit to entire long running computations, but it would also allow fresh parties to join midway through, shepherding the computation to its end. It would also reduce reliance on parties with very large computational resources, by enabling parties with low resources to contribute in long computations. This would effectively yield a *weighted*, privacy preserving, distributed computing system.

Highly dynamic computational settings have already started to appear in practice, *e.g.* Bitcoin [Nak08], Ethereum [B$^+$14], and TOR [DMS04]. These networks are powered by volunteer nodes that are free to come and go as they please, a model that has proven to be wildly successful. Designing networks to accommodate high churn rates means that anyone can participate in the protocol, no matter their computational power or availability. Building MPC protocols that are amenable to this setting would be an important step towards replicating the success of these networks. This would allow the creation of volunteer networks capable of *private computation*, creating an "MPC-as-a-service" [BHKL18] system and democratizing access to privacy preserving computation.

Fluid MPC. To bring MPC to highly dynamic settings, we formalize the study of *fluid MPC*. Consider a group of clients that wish to compute a function on confidential inputs, but do not have the resources to conduct the full computation themselves. These clients share their inputs in a privacy preserving manner with some initial *committee* of (volunteer) servers. Once the computation begins, both the clients and the initial servers may exit the protocol execution. Additionally, other servers, even those not present during the input stage, can "sign-up" to join part-way through the protocol execution, and then may later leave before the computation finishes. Informally, the work that these transient servers perform should be proportional only to a fraction of the circuit size, as they are only present for a fraction of the protocol execution. The resulting protocol should still provide the security properties we expect from MPC.

We consider a model in which the computation is divided into an *input* stage, an *execution* stage, and an *output* stage. We illustrate this in Figure 1.1. During the input stage, a set of clients prepare their inputs for computation and hand them over to the first committee of servers. The execution stage is further divided into a sequence of *epochs*. During each epoch, a committee of servers are responsible for doing some part of the computation, and then the intermediary state of the computation is securely transferred to a new committee. Critically, this work must be independent of the depth of the circuit being computed. Once the full circuit has been evaluated, there is an output stage where the final results are recovered by the clients.

In order to see how well suited a particular protocol is to this dynamic setting, we introduce the notion of *fluidity* of a protocol. Fluidity captures the minimum commitment required from each server participating in the execution stage, measured in communication rounds. More specifically, fluidity is the number of communication rounds within an epoch.

A protocol with worse fluidity might require that servers remain active to send, receive, or act as passive observers on many rounds of communication. In this sense, MPC protocols designed for static participants have the worst possible fluidity — all participants must remain active throughout the lifetime of the entire protocol. In this work, we focus on protocols with only a single round of communication per epoch, which we say achieve *maximal fluidity*. Note that such protocols must have no intra-committee communication, as the communication round must be used to transfer state.

Recall that the idea of flexibility is central to the goal of Fluid MPC. Protocols with maximal fluidity give the most flexibility to the servers participating in the protocol. It allows owners of computational resources to contribute spare cycles to MPC during downtime, and a quick exit (without disrupting computation) when they are needed for another, possibly a more important task. Moreover, since one of the motivations behind Fluid MPC is to enable long computations, we require the computation done by the servers in each epoch to be *independent* of the size of the function/circuit (or at least the depth). The goal of our work is to achieve these two properties *simultaneously.*

There are several other modeling choices that can significantly impact feasibility and efficiency of a fluid MPC protocol — many of which are non-trivial and unique to this setting. For instance: when and how are the identities of the servers in the committee of a particular epoch fixed? What requirements are there on the churn rate of the system? How does the adversary's corruption model interact with the dynamism of the protocol participants? We have already seen from the extensive literature on consensus networks that different networks make different, reasonable assumptions and arrive at very different protocols.

We discuss these modeling choices and provide a formal treatment of fluid MPC in Section 4.2. For the constructions we give in this work, we assume that the identities of the servers in a committee

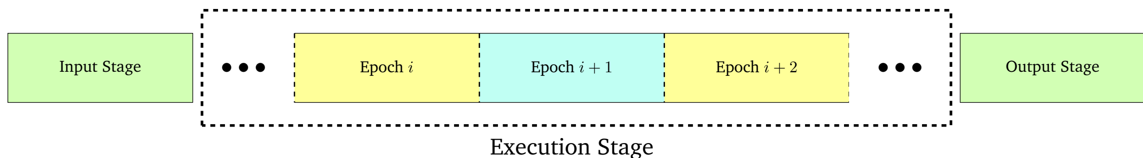


Figure 1.1: Computation model of fluid MPC.

are made known during the previous epoch.

Applications. We imagine that fluid MPC will be most useful for applications that involve long-running computations with deep circuits. In such a setting, being able to temporarily enlist dynamic computing resources could facilitate privacy-preserving computations that are difficult or impossible with limited static resources. This model would be especially valuable in scientific computing, where deep circuits are common and resources can be scarce. Consider, for example, an optimization problem with many constraints over distributed medical datasets. Using a fluid MPC protocol makes it more feasible to perform such a computation with limited resources: the privacy provided by MPC can help clear important regulatory or legal impediments that would otherwise prevent stakeholders from contributing data to the analysis, and a dynamic participation model can allow stakeholders to harness computing resources as they become available.

Prior Work: Player Replaceability. In recent years, the notion of *player replaceability* has been studied in the context of Byzantine Agreement (BA) [Mic17, CM19]. These works design BA protocols where after every round, the "current" set of players can be replaced with "new" ones without disrupting the protocol. This idea has been used in the design of blockchains such as Algorand [GHM+17a], where player replaceability helps mitigate targeted attacks on chosen participants after their identity is revealed.

Our work can be viewed as extending this line of research to the setting of MPC. We note that unlike BA where the parties have no private states – and hence, do not require state transfer for achieving player replaceability – the MPC setting necessitates a state transfer step to accommodate player churn. Maximal fluidity captures the best possible scenario where this process is performed in a *single* round.

1.2.1 Our Contributions

In this work, we initiate the study of fluid MPC. We state our contributions below.

Model. We provide a formal treatment of fluid MPC, exploring possible modeling choices in the setting of dynamic participants.

Protocols With Maximal Fluidity. We construct information-theoretic fluid MPC protocols that achieve maximal fluidity. We consider adversaries that (adaptively) corrupt any minority of the servers in each committee.

We begin by observing that the protocol by Genarro, Rabin and Rabin [GRR98], which is an optimized version of the classical semi-honest BGW protocol [BGW88] can be adapted to the fluid MPC setting in a surprisingly simple manner. We call this protocol Fluid-BGW. This protocol also achieves division of work, in the sense that the amount of work that each committee is required to do is independent of the depth of the circuit.

To achieve security against malicious adversaries, we extend the "additive attack" paradigm of [GIP+14] to the fluid MPC setting, showing that any malicious adversarial strategy on semi-honest fluid MPC protocols (with a specific structure and satisfying a weak notion of privacy against malicious adversaries[5]) is limited to injecting additive values on the intermediate wires of the circuit. We use this observation to build an *efficient* compiler (in a similar vein as recent works of [CGH+18, NV18]) that transforms such semi-honest fluid MPC protocols into ones that achieve security with abort against malicious adversaries. Our compiler enjoys two salient properties:

- It *preserves fluidity* of the underlying semi-honest protocol.
- It incurs a *multiplicative overhead of only* 2 (for circuits over large fields) in the communication complexity of the underlying protocol.

[5]It was observed in [GIP+14] that almost all known secret sharing based semi-honest protocols in the static model naturally satisfy this weak privacy property. We observe that the fluid version of BGW continues to satisfy this property. Further, we conjecture that most secret-sharing based approaches in the fluid MPC setting would also yield semi-honest protocols that achieve this property.

Applying our compiler to Fluid-BGW yields a maximally fluid MPC protocol that achieves security with abort against malicious adversaries.

We note that, while we consider a slightly restrictive setting where the adversary is limited to corrupting a minority of servers in each committee, there is evidence that our assumption might hold in practice if we, e.g., leverage certain blockchains. The work of [BGG+20] (see also [GHM+20]) explores a similar problem of dynamism in the context of secret-sharing with a similar honest-majority assumption as in our work. They show that in certain blockchain networks, it is possible to leverage the honest-majority style assumption (which is crucial to the security of such blockchains) to elect committees of servers with an honest majority of parties. The same mechanism can also be used in our work (we discuss this in more detail in Section 4.2.2). Moreover, the honest majority assumption is necessary for achieving information-theoretic security (or for using assumptions weaker than oblivious transfer), for the same reasons as in standard (static) MPC.

Implementation. We implement Fluid-BGW and our malicious compiler in `C++`, building off the code-base of [Cry19, CGH+18]. We run our implementation across multiple network settings and give concrete measurements. We discuss results from our implementation in the supplementary material (Section 4.8).

1.2.2 Related Work

Proactive Multiparty Computation. The proactive security model, first introduced in [OY91], aims to model the persistent corruption of parties in a distributed computation, and the continuous race between parties for corruption and recovery. To capture this, the model defines a "mobile" adversary that is not restricted in the total number of corruptions, but can corrupt a subset of parties in different time periods, and the parties periodically reboot to a clean state to mitigate the total number of corruptions. Prior works have investigated the feasibility of proactive security both in the context of secret sharing [HJKY95, MZW+19] and general multiparty computation [OY91, BELO14, EOPY18].

While both fluid MPC and Proactive MPC (PMPC) consider dynamic models, the motivation be-

hind the two models are completely different. This in turn leads to different modeling choices. Indeed, the dynamic model in PMPC considers slow-moving adversaries, modeling a spreading computer virus where the set of participants are fixed through the duration of the protocol. This is in contrast to the Fluid MPC model where the dynamism is derived from participants leaving and joining the protocol execution as desired. As such, the primary objective of our work is to construct protocols that have maximal fluidity while simultaneously minimizing the computational complexity in each epoch. Neither of these goals are a consideration for protocols in the PMPC setting. Furthermore, unlike PMPC, fluid MPC captures the notion of volunteer servers that sign-up for computation proportional to the computational resources available to them.

The difference in motivation highlighted above also presents different constraints in protocol design. For instance, unlike PMPC, the size of private states of parties is a key consideration in the design of fluid MPC; we discuss this further in Section 4.1. We do note, however, that some ideas from the PMPC setting, such as state re-randomization are relevant in our setting as well.

Transferable MPC. In [CH14], Clark and Hopkinson consider a notion of Transferable MPC (T-MPC) where parties compute partial outputs of their inputs and transfer these shares to other parties to continue computation while maintaining privacy. Unlike our setting, the sequence of transfers, and the computation at each step is determined completely by the circuit structure. In the constructed protocol, each partial computation involves multiple rounds of interaction and therefore does not achieve fluidity; additionally parties cannot leave during computation sacrificing on dynamism.

Concurrent and Independent Work. Two independent and concurrent works [GKM+20, BGG+20] that recently appeared on ePrint Archive also model dynamic computing environments by considering protocols that progress in discrete stages denoted as epochs, which are further divided into computation and hand-off phases. These works study and design *secret sharing* protocols in the dynamic environment. In contrast, our work focuses on the broader goal of *multi-party computation* protocols for all functionalities.

Furthermore, we focus on building protocols that achieve maximal fluidity. While this goal is not

considered in [GKM+20], [BGG+20] can be seen as achieving maximal fluidity for secret sharing. In choosing committees for each epoch, [GKM+20] consider an approach similar to ours where the committee is announced at the start of the hand-off phase of each epoch. [BGG+20] leverage properties in the blockchain to implement a committee selection procedure that ensures an honest majority in each committee.

Lastly, both of these works consider a security model incomparable to ours. Specifically, they consider security with guaranteed output delivery for secret sharing against computationally bounded adversaries, whereas we consider MPC with security with abort against computationally unbounded adversaries.

Malicious Security Compilers for MPC. There has been a recent line of exciting work [CGH+18, NV18, LN17, ABF+17, AFL+16, MRZ15, IKHC14, FL19] in designing concretely efficient compiler that upgrade security from semi-honest to malicious in the honest majority setting. Some of these compilers rely on the additive attack paradigm introduced in [GIP+14]. We take a similar approach, but adapt and extend the additive attack paradigm to the fluid MPC setting.

1.3 Bibliographic Notes

The result on order-C secure multiparty computation is based on joint work [BGJK21] with Gabrielle Beck, Abhishek Jain and Gabriel Kaptchuk that appeared at EUROCRYPT 2021 and the result on secure multiparty computation with dynamic participants is based on join work [CGG+21] with Arka Rai Choudhuri, Matthew Green, Abhishek Jain and Gabriel Kaptchuk that appeared at CRYPTO 2021.

1.4 Outline of the Thesis

In Chapter 2, we start by recalling basic definitions of secure multiparty computation and describing polynomial-based threshold and packed secret sharing schemes. In Chapter 3, we present our results

on order-C secure multiparty computation and in Chapter 4, we present our results on MPC with dynamic participants.

Chapter 2

Preliminaries

2.1 Secure Multiparty Computation

Secure multi-party computation protocol (MPC) is a protocol executed by n parties $\mathcal{P} = \{P_1, \cdots, P_n\}$ for a functionality $\mathcal{F}$. We allow for parties to exchange messages simultaneously. In every round, every party is allowed to exchange messages with other parties using different communication channels, depending on the model. A protocol is said to have k *rounds* if it proceeds in k distinct and interactive rounds.

2.1.1 Adversarial Behavior

One of the primary goals in MPC is to protect the honest parties against dishonest behavior of the corrupted parties. This is usually modeled using a central adversarial entity, that controls the set of corrupted parties and instructs them on how to operate. That is, the adversary obtains the views of the corrupted parties, consisting of their inputs, random tapes and incoming messages, and provides them with the messages that they are to send in the execution of the protocol. In our protocols we only consider the case where the adversary can only control a minority of the parties in the protocol. We discuss the following adversarial models in detail:

1. **Semi-Honest Adversaries:** A semi-honest adversary always follows the instructions of the

protocol. This is an "honest but curious" adversarial model, where the adversary might try to learn extra information by analyzing the transcript of the protocol later.

2. **Malicious Adversaries:** A malicious adversary can deviate from the protocol and instruct the corrupted parties to follow any arbitrary strategy.

We provide the basic definitions for secure multiparty computation according to the real/ideal paradigm [Gol04]. Informally, a protocol is considered secure if whatever an adversary can do in the real execution of protocol, can be done also in an ideal computation, in which an uncorrupted trusted party assists the computation.

2.1.2 Security Definitions

Real World. The real world execution of a protocol $\Pi = (P_1, \ldots, P_n)$ begins by an adversary $\mathcal{A}$ selecting any arbitrary subset of parties $\mathcal{I} \subset [n]$ to corrupt. The parties then engage in an execution of a real n-party protocol Π. Throughout the execution of Π, the adversary $\mathcal{A}$ sends all messages on behalf of the corrupted parties, and may follow an arbitrary polynomial-time strategy. In contrast, the honest parties follow the instructions of Π. At the conclusion of the protocol, each honest party outputs all the outputs it obtained in the computations. Malicious parties may output an arbitrary PPT function of the view of $\mathcal{A}$. This joint execution of Π under $(\mathcal{A}, \mathcal{I})$ in the real model, on input vector $\vec{x} = (x_1, \ldots, x_n)$, auxiliary input z and security parameter λ, denoted by $\mathsf{REAL}_{\Pi,\mathcal{I},\mathcal{A}(z)}\left(1^\lambda, \vec{x}\right)$, is defined as the output vector of $P_1, \ldots, P_n$ and $\mathcal{A}(z)$ resulting from this protocol interaction.

Ideal World. We now present standard definitions of ideal-model computations that are used to define security with abort. We start by presenting the ideal-model computation for security with abort, where the adversary may abort the computation either before or after it has learned the output; other ideal-model computations are defined either by allowing the adversary to selectively abort to some parties but not to others or by restricting the power of the adversary either by forcing the adversary to identify a corrupted party in case of abort, or no abort (guaranteed output delivery).

Ideal Computation with Abort. An ideal computation with abort of an n-party functionality $\mathcal{F}$ on input $\vec{x} = (x_1, \ldots, x_n)$ for parties $(P_1, \ldots, P_n)$ in the presence of an ideal-model adversary $\mathcal{A}$ controlling the parties indexed by $\mathcal{I} \subset [n]$, proceeds via the following steps.

Sending inputs to trusted party: For each $i \notin \mathcal{I}$, P_i sends its input x_i to the trusted party. If $i \in \mathcal{I}$, the adversary may send to the trusted party any arbitrary input for the corrupted party P_i. Let x'_i be the value actually sent as the ith party's input.

Early abort: The adversary $\mathcal{A}$ can abort the computation by sending an abort message to the trusted party. In case of such an abort, the trusted party sends $\perp$ to all parties and halts.

Trusted party answers adversary: The trusted party computes $(y_1, \ldots, y_n) = \mathcal{F}(x'_1, \ldots, x'_n)$ and sends y_i to party P_i for every $i \in \mathcal{I}$.

Late abort: The adversary $\mathcal{A}$ can abort the computation (after seeing the outputs of corrupted parties) by sending an abort message to the trusted party. In case of such abort, the trusted party sends $\perp$ to all honest parties and halts. Otherwise, the adversary sends a continue message to the trusted party.

Trusted party answers remaining parties: The trusted party sends y_i to P_i for every $i \notin \mathcal{I}$.

Outputs: Honest parties always output the message received from the trusted party and the corrupted parties output nothing. The adversary $\mathcal{A}$ outputs an arbitrary function of the initial inputs x_i s.t. $i \in \mathcal{I}$, the messages received by the corrupted parties from the trusted party and its auxiliary input.

Definition 1 (Ideal-model computation). *Let $\mathcal{F} : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be an n-party functionality. let $\mathcal{I} \subset [n]$ be the set of indices of the corrupted parties, and let λ be the security parameter. Then, the joint execution of $\mathcal{F}$ under $(\mathcal{A}, \mathcal{I})$ in the ideal model, on input vector $\vec{x} = (x_1, \ldots, x_n)$, auxiliary input z to $\mathcal{A}$ and security parameter λ, denoted $\mathsf{IDEAL}_{\mathcal{F},\mathcal{I},\mathcal{A}(z)}(1^\lambda, \vec{x})$, is defined as the output vector of $P_1, \ldots, P_n$ and $\mathcal{A}$ resulting from the above described ideal process.*

Security Having defined the real and ideal models, we can now define security of protocols according to the real/ideal paradigm. Since we work in the information-theoretic setting, we only give a definition for statistically secure protocols.

Definition 2. *Let $\mathcal{F} : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be an n-party functionality and let Π be a probabilistic polynomial-time protocol computing $\mathcal{F}$. The protocol Π computes $\mathcal{F}$ with statistical security against at most t corruptions with abort, if for every unbounded real-model adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ for the ideal model, who's running time is polynomial in the running time of $\mathcal{A}$, such that for every $\mathcal{I} \subset [n]$ of size at most t, it holds that*

$$\left\{\mathsf{REAL}_{\Pi,\mathcal{I},\mathcal{A}(z)}\left(1^\lambda, \vec{x}\right)\right\}_{(\vec{x},z)\in(\{0,1\}^*)^{n+1},\lambda\in\mathbb{N}} \approx_s \left\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{I},\mathcal{S}(z)}(1^\lambda, \vec{x})\right\}_{(\vec{x},z)\in(\{0,1\}^*)^{n+1},\lambda\in\mathbb{N}}$$

2.2 Secret Sharing

2.2.1 Threshold Secret Sharing

A t-out-of-n secret sharing scheme enables n parties to share as secret $v \in \mathbb{F}$ so that no subset of t parties can learn any information about it, while any subset of $t+1$ parties can reconstruct it. We use *Shamir's secret sharing* scheme [Sha79a] in our protocols that supports the following procedures[1]:

– $\mathsf{share}(v, t+\ell)$: In this procedure, a dealer shares a value $v \in \mathbb{F}$ as follows:

1. Set $p_0 = v$ and sample a random polynomial $q(z)$ of degree t such that $q(0) = 0$.
2. Set $p(z) = p_0 + q(z)\prod_{i=1}^{\ell}(z - e_i)$, where $e_1, \ldots, e_\ell$ are preselected elements in $\mathbb{F}$.
3. For each $i \in [n]$, set $v_i = p(i)$.

Each output share v_i (for $i \in [n]$) is the share intended for party P_i. We denote the $t+\ell$-out-of-n sharing of a value v by $[v]$. We use the notation $[v]_J$ to denote the shares held by a subset of parties $J \subset [n]$. We stress that if the dealer is corrupted, then the shares received by the parties

[1] We note that this is a generalized version of the traditional Shamir secret sharing scheme (This is necessary for our application in Chapter 3). In particular, the traditional version can be derived by setting $\ell = 0$. In Chapter 4, we will work with the traditional version and assume $\ell = 0$

may not be correct. Nevertheless, we abuse notation and say that the parties hold shares $[v]$ even if these are not correct.

- share$(v, J, [v]_J, t+\ell)$: This procedure is similar to the previous procedure, except that here the shares of a subset J of parties with $|J| \le t$ are fixed in advance. Given the value v to be shared, let $p(z) = v + p_1 z + p_2 z^2 + \ldots + p_t z^{t+\ell}$ be the polynomial used for secret sharing. Now given $|J|$ shares, we get the following system of equations:

$$\forall i \in J,\ v_i = v + p_1 i + p_2 i^2 + \ldots + p_t i^t$$

  This a system of $|J|$ equations in t variables $\{p_1, \ldots, p_t\}$ and can be easily solved using Gaussian elimination. Finally, given the polynomial $p(z)$ the shares of all other parties $i \in [n] \setminus J$ is $v_i = p(i)$.

Computation Cost of Secret Sharing. Naively computing shares of a secret requires the dealer to evaluate a polynomial of degree $t+\ell$ on n distinct points. This incurs a total computation complexity of $O(n^2)$. An optimized implementation using Fast Fourier Transform (FFT) can be used to improve the computation complexity to $O(n \log n)$. Whereas the total communication cost of sending these shares to the respective parties is only $O(n)$. As a result, the total computation complexity of MPC protocols based on threshold secret sharing is typically more than the communication complexity.

2.2.2 Packed Secret Sharing

A packed secret sharing scheme enables n parties to share a block of ℓ secrets $v = (s_1, \ldots, s_\ell) \in \mathbb{F}^\ell$ so that no subset of at most $t - \ell + 1$ parties can learn any information about it, while any subset of $D + 1$ parties can reconstruct it (for application in Chapter 3, we assume $D = t + 2\ell - 1$). We use the multi-secret generalization of Shamir's secret sharing scheme as introduced by Franklin et. al [FY92]. Let $\alpha_1, \ldots, \alpha_n$ and $e_1, \ldots, e_\ell$ be $n + \ell$ preselected elements in $\mathbb{F}$ that are known to all parties. This packed secret sharing scheme supports the following procedures:

- $\mathsf{pshare}(s, D)$: In this procedure, a dealer shares a block of ℓ secrets $s = (s_1, \ldots, s_\ell) \in \mathbb{F}^\ell$ using a random polynomial $p(z)$ of degree D over $\mathbb{F}$, subject to the constraint $p(e_i) = s_i$ for each $1 \leq i \leq \ell$. This is done as follows:

  1. Pick a random polynomial $q(z)$ of degree $D - \ell$.
  2. Set
  $$p(z) = q(z) \prod_{i=1}^{\ell} (x - e_i) + \sum_{i=1}^{\ell} s_i L_i(z),$$
  where $L_i(z)$ is the Lagrange polynomial $\frac{\Pi_{j \neq i}(x - e_j)}{\Pi_{j \neq i}(e_i - e_j)}$.
  3. For each $i \in [n]$, send $p(\alpha_i)$ to party P_i.

- $\mathsf{pshare}(v, J, [v]_J, D)$: This procedure is similar to the packed secret sharing procedure using a univariate polynomial, except that here the shares of a subset J of parties with $|J| \leq D$ are fixed in advance. Given a block of ρ values $v = (s_1, \ldots, s_\rho)$ to be shared, let $p(z) = p_0 + p_1 z + p_2 z^2 + \ldots + p_{t+\rho} z^{t+\rho}$ be the polynomial used for secret sharing. Now given $|J|$ shares and ρ secret values $(s_1, \ldots, s_\rho)$, we get the following system of equations:

  $$\forall i \in J,\ v_i = p_0 + p_1 i + p_2 i^2 + \ldots + p_{t+\rho} i^{t+\rho}$$

  $$\forall i \in [\rho],\ v_i = p_0 + p_1 \mu_i + p_2 \mu_i^2 + \ldots + p_{t+\rho} \mu_i^{t+\rho}$$

  This a system of $|J| + \rho$ equations in $t + \rho + 1$ variables $\{p_0, \ldots, p_{t+\rho}\}$ and can be easily solved using Gaussian elimination. Finally, given the polynomial $p(z)$ the shares of all other parties $i \in [n] \setminus J$ is $v_i = p(i)$.

Computation Cost of Packed Secret Sharing. Let ℓ be a constant fraction of n, then the naive cost of computing packed shares of a secret vector of length ℓ is $O(n^2)$. As before, using FFT can bring this cost down to $O(n \log n)$, i.e., the amortized cost of sharing a single element in the vector is $O(\log n)$. The amortized communication cost associated with sharing a single element of the vector is $O(1)$. As a result, the total computation complexity of MPC protocols based on packed secret sharing is also typically more than the communication complexity.

Chapter 3

Order-C Secure Mutliparty Computation

3.1 Technical Overview

We begin our technical overview by recalling the key techniques developed in prior works for reducing dependence on the number of parties. We then proceed to describe our main ideas in Section 3.1.2.

3.1.1 Background

Classical MPC protocols have total complexity $O(n^2|C|)$. These protocols, exemplified by [BGW88], leverage Shamir's secret sharing [Sha79b] to facilitate distributed computation and require communication for each multiplication gate to enable degree reduction. Typical multiplication subprotocols require that each party send a message to every other party for every multiplication gate, resulting in total communication complexity $O(n^2|C|)$. As mentioned earlier, two different techniques have been developed to reduce the asymptotic complexity of MPC protocols down to $O(n|C|)$: efficient multiplication techniques and packed secret sharing.

Efficient Multiplication. In [DN07], Damgård and Nielsen develop a randomness generation tech-

nique that allows for a more efficient multiplication subprotocol. At the beginning of the protocol, the parties generate shares of random values, planning to use one of these values for each multiplication gate. These shares are generated in *batches*, using a subprotocol requiring $O(n^2)$ communication that outputs $\Theta(n)$ shares of random values. This *batched randomness generation* subprotocol can be used to compute $O(|C|)$ shared values with total complexity $O(n|C|)$. After locally evaluating a multiplication gate, the players use one of these shared random values to mask the gate output. Players then send the masked gate output to a *leader*, who reconstructs and broadcasts the result back to all players.[1] Finally, players locally remove the mask to get a shared value of the appropriate degree. This multiplication subprotocol has complexity $O(n)$.

Packed Secret Sharing. In [FY92], Franklin and Yung proposed a vectorized version of Shamir secret sharing called *packed secret sharing* that trades a lower corruption threshold for more efficient representation of secrets. More specifically, their scheme allows a dealer to share a vector of $\Theta(n)$ secrets such that each of the n players still only hold a single field element. Importantly, the resulting shares preserve a SIMD version of the homomorphisms required to run MPC. Specifically, if $X = (x_1, x_2, x_3)$ and $Y = (y_1, y_2, y_3)$ are the vectors that are shared and added or multiplied, the result is a sharing of $X + Y = (x_1 + y_1, x_2 + y_2, x_3 + y_3)$ or $XY = (x_1y_1, x_2y_2, x_3y_3)$ respectively. Like traditional Shamir secret sharing, the degree of the polynomial corresponding to XY is twice that of original packed sharings of X and Y. This allows players to compute over $O(n)$ gates *simultaneously*, provided two properties are satisfied: (1) all of the gates *perform the same operation* and (2) the inputs to each gate *are in identical positions in the respective vectors*. In particular, it is not possible to compute x_1y_2 in the previous example, as x_1 and y_2 are not *aligned*. However, if the circuit has the correct structure, packed secret sharing reduces MPC complexity from $O(n^2|C|)$ to $O(n|C|)$.

3.1.2 Our Approach: Semi-Honest Security

A Strawman Protocol. A natural idea towards achieving $O(|C|)$ MPC is to design a protocol that can take advantage of *both* efficient multiplications and packed secret sharing. As each technique

[1] The choice of the leader can be rotated amongst the players to divide the total computation.

asymptotically shaves off a factor of n, we can expect the resulting protocol to have complexity $O(|C|)$. A naïve (strawman) protocol combining these techniques might proceed as follows:

– Players engage in a first phase to generate packed shares of random vectors using the batching technique discussed earlier. This subprotocol requires $O(n^2)$ messages to generate $\Theta(n)$ shares of packed random values, each containing $\Theta(n)$ elements. As we need a single random value per multiplication gate, $O(|C|)$ total messages are sent.

– During the input sharing phase, players generate packed shares of their inputs, distributing shares to all players.

– Players proceed to evaluate the circuit over these packed shares, using a single leader to run the efficient multiplication protocol to reduce the degrees of sharings after multiplication. This multiplication subprotocol requires $O(n)$ communication to evaluate $\Theta(n)$ gates, so the total complexity is $O(|C|)$.

– Once the outputs have been computed, players broadcast their output shares and reconstruct the output.

While natural, this template falls short because the circuit may not satisfy the requirements to perform SIMD computation over packed shares. As mentioned before, packed secret sharing only offers savings if all the simultaneously evaluated gates are the same and all gate inputs are properly aligned. However, this is an unreasonable restriction to impose on the circuits. Indeed, running into this problem, [DIK10, GIP15] show that any circuit can be modified to overcome these limitations, at the cost of a significant blowup in the circuit size, which adversely affects their computation and communication efficiency. (We discuss their approach in more detail later in this section.)

Our Ideas. Without such a circuit transformation, however, it is not immediately clear how to take advantage of packed secret sharing (other than for SIMD circuits). To address this challenge, we devise two conceptual tools, each of which we will "simulate" using existing primitives, as described below:

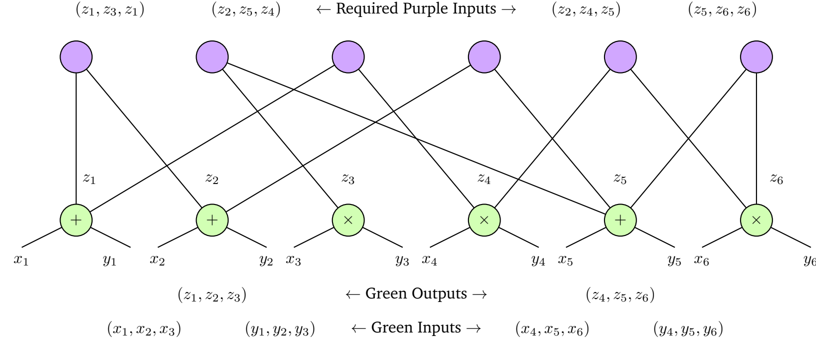


Figure 3.1: A simple example pair of circuit layers illustrating the need for differing-operation packed secret sharing and our realignment procedure.

1. *Differing-operation packed secret sharing*, a variant of packet secret sharing in which different operations can be evaluated for each position in the vector. For example, players holding shares of (x_1, x_2, x_3) and (y_1, y_2, y_3) are unable to compute $(x_1y_1, x_2+y_2, x_3y_3)$. With *differing-operation packed secret sharing*, we imagine the players can generate an operation vector (*e.g.* $(\times, +, \times)$) and apply the corresponding operation to each pair of inputs. Given such a primitive, there would be no need to modify a circuit to ensure that shares are evaluated on the same kind of gate.

2. *A realignment procedure* that allows pre-existing packed secret shares to be modified so previously unaligned vector entries can be moved and aligned properly for continued computation without requiring circuit modification.

We note that highly repetitive circuits are *layered* circuits (that is the inputs to layer $i+1$ of a circuit are all output wires from layer i). For the remainder of this section, we will make the simplifying assumption that circuits contain only multiplication and addition gates and that the circuit is layered. We expand our analysis to cover other gates (e.g. relay gates) in the technical sections.

Simulating Differing-operation Packed Secret Sharing. To realize differing-operation packed secret sharing, we require the parties to compute *both* operations over their input vectors. For instance, if the player hold share of (x_1, x_2, x_3) and (y_1, y_2, y_3) and wish to compute the operation vector

$(\times, +, \times)$, they begin by computing both $(x_1 + y_1, x_2 + y_2, x_3 + y_3)$ and (x_1y_1, x_2y_2, x_3y_3). Note that all the entries required for the final result are contained in these vectors, and the players just need to "select" which of the aligned entries will be included in the final result.

Recall that in the multiplication procedure described earlier, the leader reconstructs all masked outputs before resharing them. We modify this procedure to have the leader reconstruct both the sum and product of the input vectors, *i.e.* the unpacked values $x_1+y_1, x_2+y_2, x_3+y_3, x_1y_1, x_2y_2, x_3y_3$ (while masked). The leader then performs this "selection" process, and packs only the required values to get a vector $(x_1y_1, x_2 + y_2, x_3y_3)$, and discards the unused values $x_1 + y_1, x_2y_2, x_3 + y_3$. Shares of this vector are then distributed to the rest of the players, who unmask their shares. Note that this procedure only has an overhead of 2, as both multiplication and addition must be computed.[2]

Simulating the Realignment Procedure. First note that realigning packed shares may require not only internal permutations of the shares, but also swapping values across vectors. For example, consider the circuit snippet depicted in Figure 3.1.2. The outputs of the green (bottom) layer are not structured correctly to enable computing the purple (top) layer, and require this cross-vector swapping. As such, we require a realignment procedure that takes in all the vectors output by computing a particular circuit layer and outputs multiple properly aligned vectors.

Our realignment procedure builds on the ideas used to realize differing-operation packed secret sharing. Recall that the leader is responsible for reconstructing the masked result values from *all* gates in the previous layer. With access to all these masked values, the leader is not only able to select between a pair of values for each element of a vector (as before), but instead can arbitrarily select the values required from across all outputs. For instance, in the circuit snippet in Figure 3.1.2, the leader has masked, reconstructed values $z_i^{\mathrm{add}}, z_i^{\mathrm{mult}}$ for $i \in [6]$. Proceeding from left to right of the purple layer, the leader puts the value corresponding to the left input wire of a gate into a vector and the right input wire value into the correctly aligned slot of a corresponding vector. Using this procedure, the input vectors for the first three gates of the purple layer will be $(z_1^{\mathrm{add}}, z_3^{\mathrm{mult}}, z_1^{\mathrm{add}})$ (left wires) and $(z_2^{\mathrm{add}}, z_5^{\mathrm{add}}, z_4^{\mathrm{mult}})$ (right wires).

[2] In this toy example only one vector is distributed back to the parties. If layers are approximately of the same size, an approximately equal number of vectors will be returned.

Putting it Together. We are now able to refine the strawman protocol into a functional protocol. When evaluating a circuit layer, the players run a protocol to simulate differing-operation packed secret sharing, by evaluating each gate as both an addition gate and multiplication gate. Then, the leader runs the realignment procedure to prepare vectors that are appropriate for the next layer of computation. Finally, the leader secret shares these new vectors, distributing them to all players, and computing the next layer can commence. Conceptually, the protocol uses the leader to "unpack" and "repack" the shares to simultaneously satisfy both requirements of SIMD computation.

Leveraging Circuits with Highly Repetitive Structure. Until this point, we have been using the masking primitive imprecisely, assuming that it could accommodate the procedural changes discussed above without modification. This however, is not the case. Because we need to mask and unmask values while they are in a packed form, *the masks themselves must be generated and handled in packed form*.

Consider the example vectors used to describe differing-operation packed secret sharing, trying to compute $(x_1y_1, x_2+y_2, x_3y_3)$ given (x_1, x_2, x_3) and (y_1, y_2, y_3). If the same mask (r_1, r_2, r_3) is used to mask both the sum and product of these vectors, privacy will not hold; for example, the leader will open the values $x_1 + y_1 + r_1$ and $x_1y_1 + r_1$, and thus learn something about x_1 and y_1. If (r_1, r_2, r_3) is used to mask addition and (r'_1, r'_2, r'_3) is used for multiplication, there is privacy, but it is unclear how to unmask the result. The shared vector distributed by the leader will correspond to $(x_1y_1 + r_1, x_2 + y_2 + r'_2, x_3y_3 + r_3)$ and the random values cannot be removed with only access to (r_1, r_2, r_3) and (r'_1, r'_2, r'_3). To run the realignment procedure, the same problem arises: the unmasking vectors must have a different structure than the masking vectors, with their relationship determined by the structure of the next circuit layer.

We overcome this problem by making modifications to the batched randomness generation procedure. Instead of generating structurally identical masking and unmasking shares, we instead use the circuit structure to permute the random inputs used during randomness generation so we get outputs of the right form. In the example above, the players will collectively generate the *masking vectors* (r_1, r_2, r_3) and (r'_1, r'_2, r'_3), where each entry is sampled independently at random. The

players then generate the *unmasking vector* (r_1, r_2', r_3) by permuting their inputs to the generation algorithm. For a more complete description of this subprotocol, see Section 3.5.1.

However, recall that it is critical for efficiency that we generate all randomness in *batches*. By permuting the inputs to the randomness generation algorithm, we get $\Theta(n)$ masks that are correctly structured *for a particular part of the circuit structure*. If this particular structure occurs only once in the circuit, only one of the $\Theta(n)$ shares can actually be used during circuit evaluation. In the worst case, if each circuit substructure is *unique*, the resulting randomness generation phase requires $O(n|C|)$ communication complexity.

This is where the requirement for highly repetitive circuits becomes relevant. This class of circuits guarantees that (1) the circuit layers are wide enough that using packed secret sharing with vectors containing $\Theta(n)$ elements is appropriate, and (2) all $\Theta(n)$ shares of random values generated during the batched randomness generation phase can be used during circuit evaluation. We note that this is a rather simplified version of the definition, we give a formal definition of such circuits in Section 3.3.2.

Non-interactive packed secret sharing from traditional secret shares. Another limitation of the strawman protocol presented above is that the circuit must ensure that all inputs from a single party can be packed into a single packed secret sharing at the beginning of the protocol. We devise a novel strategy that allows parties to secret share each of their inputs individually using regular secret sharing. Parties can then *non-interactively* pack the appropriate inputs according to the circuit structure. This strategy can also be used to efficiently *switch* to protocols $O(n|C|)$ protocols when parts of the circuit lack highly repetitive structure; the leader omits the repacking step, and the parties compute on traditional secret share until the circuits becomes highly repetitive, at which point they non-interactively re-packing any wire values (see Section 3.3.4).

Existing $O(|C|)$ protocols like [DIK10] do not explicitly discuss how their protocol handles this input scenario. We posit that this is because there are generic transformations like embedding switching networks at the bottom of the circuit that allow any circuit to be transformed into a circuit in which a player's inputs can be packed together. Unsurprisingly, these transformations significantly

increase the size of the circuit. Since [DIK10] is primarily concerned with asymptotic efficiency, such circuit modification strategies are sufficient for their work.

Comparison with [DIK10]. We briefly recall the strategy used in [DIK10], in order to overcome the limitations of working with packed secret sharing that we discussed earlier. They present a generic transformation that transforms any circuit into a circuit that satisfies the following properties:

1. The transformed circuit is layered and each layer only consists of one type of gates.
2. The transformed circuit is such that, when evaluating it over packed secret shares, there is never a need to permute values across different vectors/blocks that are secret shared. While the values within a vector might need to be permuted during circuit evaluation, the transformed circuit has a nice property that only $\log \ell$ (where ℓ is the size of the block) such permutations are needed throughout the circuit.

It is clear that the first property already gets around the first limitation of packed secret sharing. The second property partly resolves the *realignment* requirement from a packed secret sharing scheme by only requiring permutations within a given vector. This is handled in their protocol by generating permuted random blocks that are used for masking and unmasking in the multiplication sub-protocol. Since only $\log \ell$ different permutations are required throughout the protocol, they are able to get significant savings by generating random pairs corresponding to the same permutation in *batches*. Our "unpacking" and "repacking" approach can be viewed as a generalization of their technique, in the sense that we enable permutation and duplication of values across different vectors by evaluating the entire layer in one shot.

As noted earlier, this transformation introduces significant overhead to the size of the circuit, and is the primary reason for the large multiplicative and additive terms in the overall complexity of their protocol. As such, it is unclear how to directly use their protocol to compute circuits with highly repetitive structures, while skipping this circuit transformation step. This is primarily because these circuits might not satisfy the first property of the transformed circuit. Moreover, while it is true that the number of possible permutations required in such circuits are very few, they might require

permuting values across different vectors, which cannot be handled in their protocol.

3.1.3 Malicious Security

Significant work has been done in recent years to build compilers that take semi-honest protocols that satisfy common structures and produce efficient malicious protocols, most notably in the "additive attack paradigm" described in [GIP+14]. These semi-honest protocols are secure *up to additive attacks*, that is any adversarial strategy is only limited to injecting additive errors onto each of the wires in the circuit that are independent of the "actual"wire values. The current generation of compilers for this class of semi-honest protocols, exemplified by [CGH+18, NV18, FL19, GSZ20], introduce only a small multiplicative overhead (e.g., 2 in the case of [CGH+18]) and require only a constant number of additional rounds to perform a single, consolidated check

Genkin et al. showed in [GIP15] (with additional technical details in [Gen16]) that protocols leveraging packed secret sharing schemes do not satisfy the structure required to leverage the compilers designed in the "additive attack paradigm." Instead, they show that most semi-honest protocols that use packed secret sharing are secure up to linear errors, that is the adversary can inject errors onto the output wires of multiplication gates that are *linear functions* of the values contained in the packed sharing of input wires to this gate. We observe that this also holds true for our semi-honest protocol. They present a malicious security compiler for such protocols that introduces a small multiplicative overhead.

To achieve malicious security, we add a new consolidated check onto our semi-honest protocol, reminiscent of the check for circuits over small-fields presented in Section 5 of [CGH+18]. The resulting maliciously secure protocol has approximately 2.3 times the complexity of our semi-honest protocol (depending on the choice of ϵ), plus a constant sized, consolidated check at the end – for the first time matching the efficiency of the compilers designed for protocols secure up to additive attacks.

As in [CGH+18], we run two parallel executions of the circuit, maintaining the invariant that for each packed set of wires $z = (z_1, z_2, \ldots, z_\ell)$ in C the parties also compute $z' = rz = (rz_1, rz_2, \ldots, rz_\ell)$

for a global, secret scalar value r. Once the players have shares of both z and z' for each wire in the circuit, we generate shares of random vectors $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_\ell)$ (one for each packed sharing vector in the protocol) using a malicious secure sub-protocol and reconstruct the value r. The parties then interactively verify that $r * \alpha * z = \alpha * z'$. Importantly, this check can be carried out simultaneously for all packed wires in the circuit, *i.e.*

$$r * \sum_{i \in C} \alpha_i * z_i = \sum_{i \in C} \alpha_i * z'_i$$

This simplified check relies heavily on the malicious security of the randomness generation sub-protocol. Because of the structure of linear attacks and the fact that α was honestly secret-shared, multiplying z and z' with α injects linear errors chosen by the adversary that are monomials in α only. That is, the equation becomes

$$r * \sum_{i \in C} (\alpha_i * z_i + E(\alpha)) = \sum_{i \in C} (\alpha_i * z'_i + E'(\alpha))$$

for adversarially chosen linear functions E and E'. Because α is independent of r and r is applied to the left hand side of this equation only at the end, this check will only pass if $r * E(\alpha) = E'(\alpha)$. For any functions $E(\cdot), E'(\cdot)$ this only happen if either (1) both are the zero function (in which case there are no errors), or (2) with probability $\frac{1}{|\mathbb{F}|}$. Hence, this technique can also be used with packed secret sharing to get an efficient malicious security compiler.

3.2 Preliminaries

Model and Notation. We consider a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$ in which each party provides inputs to the functionality, participates in the evaluation protocol, and receives an output. We denote an arbitrarily chosen special party P_{leader} for each layer (of the circuit) who will have a special role in the protocol; we note that the choice of P_{leader} may change in each layer to better distribute computation and communication. Each pair of parties are able to communicate over point-to-point

private channels.

We consider a functionality that is represented as an arithmetic circuit C over a field $\mathbb{F}$, with maximum width w and total total depth d. We visualize the circuits in a bottom-up setting (like in Merkle trees), where the input gates are at the bottom of the circuit and the output gates are at the top. As we will see later in the definition of highly repetitive circuits, we work with *layered circuits*, which comprise of layers such that the output of layer i are only used as input for the gates in layer $i+1$.

We consider security against a static adversary Adv that corrupts $t \leq n(\frac{1}{2} - \frac{2}{\epsilon})$ players, where ϵ is a tunable parameter of the system. As we will be working with both a packed secret sharing scheme (see Section 2.2.2) and a slightly modified version of regular threshold secret sharing scheme (see Section 2.2.1), we require additional notation. We denote the packing constant for our protocol as $\ell = \frac{n}{\epsilon}$. Additionally, we will denote the threshold of our packed secret sharing scheme as $D = t + 2\ell - 1$. We will denote vectors of packed values with **bold** alphabets, for instance $\mathbf{x}$. Packed secret shares of a vector $\mathbf{x}$ with respect to degree D are denoted $[\mathbf{x}]$ and with respect to degree $n-1$ as $\langle \mathbf{x} \rangle$. We let $e_1, \ldots, e_\ell$ be the fixed x-coordinates on the polynomial used for packed secret sharing, where the ℓ secrets will be stored, and $\alpha_1, \ldots \alpha_n$ be the fixed x-coordinates corresponding to the shares of the parties. For regular threshold secret sharing, we will only require shares w.r.t. degree $t + \ell$. We use the *square bracket* notation to denote a secret sharing w.r.t. degree $t + \ell$. We note that we work with a slightly modified sharing algorithm of the Shamir's secret sharing scheme (see Section 2.2.1 for details). We denote the Vandermonde matrix $\mathbf{V}_{n,(n-t)} \in \mathbb{F}^{n \times (n-t)}$. which is defined as follows:

$$\begin{bmatrix} 1 & \gamma_1 & \ldots & \gamma_1^{n-t-1} \\ 1 & \gamma_2 & \ldots & \gamma_2^{n-t-1} \\ . & . & \ldots & . \\ . & . & \ldots & . \\ 1 & \gamma_n & \ldots & \gamma_n^{n-t-1} \end{bmatrix}$$

where $\gamma_1, \ldots, \gamma_n \in \mathbb{F}$ are n distinct non-zero elements. In some cases, we also use a hyper-invertible matrix as defined in [BTH08] and denote it by $\mathbf{H}_{n,n} \in \mathbb{F}^{n \times n}$.

Chapter Organization In Section 3.3 we define the class of highly repetitive circuits and give some natural examples of such circuits. Section 3.4.3, we describe our non-interactive protocol for packing regular shares. Section 3.7 gives a construction of our semi-honest and maliciously secure protocols. In Section 3.9, we give details of our implementation and present an extensive comparison with prior work.

3.3 Highly Repetitive Circuits

In this section, we formalize the class of highly repetitive circuits and discuss some examples of naturally occurring highly repetitive circuits.

3.3.1 Wire Configuration

We start by formally defining a *gate block*, which is the minimum unit over which we will reason.

Definition 3 (Gate Block). *We call a set of j gates that are all on the same layer, a gate block. We say the size of a gate block is j.*

An additional non-standard functionality we require is an explicit wire mapping function. Recall from the technical overview that the leader must repack values according to the structure of the next layer. To reason formally over this procedure, we define the function $\mathsf{WireConfiguration}$, which takes in two blocks of gates block_{m+1} and block_m, such that the output wires of the gates in block_m feed as input to the gates in block_{m+1}. $\mathsf{WireConfiguration}$ outputs two ordered arrays $\mathsf{LeftInputs}$ and $\mathsf{RightInputs}$ that contain the indices corresponding to the left input and right input of each gate in block_{m+1} respectively. In general, we can say that $\mathsf{WireConfiguration}(\mathsf{block}_{m+1}, \mathsf{block}_m)$ will output a correct alignment for block_{m+1}. This is because for all values $j \in [|\mathsf{block}_{m+1}|]$, if the values corresponding to the wire $\mathsf{LeftInputs}[j]$ and $\mathsf{RightInputs}[j]$ are aligned, then computing block_{m+1} is possible. We describe the functionality for $\mathsf{WireConfiguration}$ in Figure 3.2. It is easy to see that the blocks $\mathsf{block}_{m+1}, \mathsf{block}_m$ must lie on consecutive layers in the circuit. We say that a pair of gate blocks is *equivalent* to another pair of gate blocks, if the outcome of $\mathsf{WireConfiguration}$ on both pairs

is identical.

The Function $\mathsf{WireConfiguration}(\mathsf{block}_{m+1}, \mathsf{block}_m)$

1. Initialize two ordered arrays $\mathsf{LeftInputs} = [\,]$ and $\mathsf{RightInputs} = [\,]$, each with capacity $|\mathsf{block}_{m+1}|$.
2. For a gate g, let $l(g) = (j, \mathsf{type})$ denote the index j and type of the gate in block block_m that feeds the left input of g. Similarly, let $r(g) = (j, \mathsf{type})$ denote the right input gate index and type of g. For gates with fan-in one, *i.e.* relay gates, $r(g) = 0$. For each gate g_j in block_{m+1}, we set
   - $\mathsf{LeftInputs}[j] = l(g_j)$
   - $\mathsf{RightInputs}[j] = r(g_j)$
3. Output $\mathsf{LeftInputs}, \mathsf{RightInputs}$.

Figure 3.2: The function $\mathsf{WireConfiguration}(\mathsf{block}_{m+1}, \mathsf{block}_m)$ that computes a proper alignment for computing block_{m+1}

3.3.2 (A, B)-Repetitive Circuits

With notation firmly in hand, we can now formalize the class of (A, B)-repetitive circuits, where A, B are the parameters that we explain next. Highly repetitive circuits are a subset of (A, B)-repetitive circuits, which we will define later.

We define an (A, B)-repetitive circuit using a partition function part that decomposes the circuit into blocks of gates, where a block consists of gates on the same layer. Let $\{\mathsf{block}_{m,j}\}$ be the output of this partition function, where m indicates the layer of the circuit corresponding to the block and j is its index within layer m. Informally speaking, an (A, B)-repetitive circuit is one that satisfies the following properties:

1. Each block $\mathsf{block}_{m,j}$ consist of at least A gates.
2. For each pair $(\mathsf{block}_{m,j}, \mathsf{block}_{m+1,j})$, all the gates in $\mathsf{block}_{m+1,j}$ only take in wires that are output wires of gates in $\mathsf{block}_{m,j}$. And the output wires of all the gates in $\mathsf{block}_{m,j}$ only go an input to the gates in $\mathsf{block}_{m+1,j}$.
3. For each pair $(\mathsf{block}_{m,j}, \mathsf{block}_{m+1,j})$, there exist at least B other pairs with identical wiring

between the two blocks.

We now give a formal definition.

Definition 4 ((A,B)-Repetitive Circuits). *We say that a layered circuit C with depth d is called an (A,B)-repetitive circuit if there exists a value $\sigma \geq 1$ and a partition function* part *which on input* layer_m *(m^{th} layer in C), outputs disjoint blocks of the form*

$$\{\mathsf{block}_{m,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m, \mathsf{layer_m}),$$

such that the following holds, for each $m \in [d], j \in [\sigma]$:

1. ***Minimum Width:*** *Each* $\mathsf{block}_{m,j}$ *consists of at least A gates.*
2. ***Bijective Mapping:*** *All the gates in* $\mathsf{block}_{m,j}$ *only take inputs from the gates in* $\mathsf{block}_{m-1,j}$ *and only give outputs to gates in* $\mathsf{block}_{m+1,j}$.
3. ***Minimum Repetition:*** *For each* $(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$*, there exist pairs $(m_1,j_1) \neq (m_2,j_2) \neq \ldots \neq (m_B,j_B) \neq (m,j)$ such that for each $i \in [B]$,* $\mathsf{WireConfiguration}(\mathsf{block}_{m_i+1,j_i}, \mathsf{block}_{m_i,j_i}) = \mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$.

Intuitively, this says that a circuit is built from an arbitrary number of gate blocks with sufficient size, and that all blocks are repeated often throughout the circuit. Unlike the layer focused example in the introduction, this definition allows layers to comprise of multiple blocks. In fact, these blocks can even *interact* by sharing input values. The limitation of this interaction, captured by the WireConfiguration check, is that the interacting inputs must come from predictable indices in the previous layer and must have the same gate type.

We also consider a relaxed variant of (A,B)-repetitive circuits, which we call (A,B,C,D)-repetitive circuits. These circuits differ from (A,B)-repetitive circuits in that they allow for a relaxation of the minimum width and repetition requirement. In particular, in an (A,B,C,D)-repetitive circuit, it suffices for all but C blocks to satisfy the minimum width requirement and similarly, all

but D blocks are required to satisfy the minimum repetition requirement. In this work, we focus on the following kind of (A, B, C, D)-repetitive circuits.

Definition 5 (Highly Repetitive Circuits). *We say that (A, B, C, D)-repetitive circuits are highly repetitive w.r.t. n parties, if $A, B \in \Omega(n)$ and C, D are some constants.*

We note that defining a class of circuits w.r.t. to the number of parties that will evaluate the circuit might a priori seem unusual. However, this is common throughout the literature attempting to achieve $O(|C|)$ MPC that use packed secret sharing. For example, the protocols in [DIK$^+$08, DIK10, GIP15] achieve $\tilde{O}(|C|)$ communication for circuits that are $\Omega(n)$ gates wide. Similarly, our work achieves $O(|C|)$ communication and computation for circuits that are $(\Omega(n), \Omega(n), C, D)$-repetitive, where C and D are constants. Alternatively, if the number of input wires are equal to the number of participating parties, we can re-phrase the above definition w.r.t. the number of input wires in a circuit.

It might be useful to see the above definition as putting a limit on the number of parties for which a circuit is highly repetitive: any (A, B, C, D)-repetitive circuit, is highly repetitive for upto $\texttt{min}(O(A), O(B))$ parties. While our MPC protocol can work for any (A, B, C, D)-repetitive circuit, it has $O(|C|)$ complexity only for highly repetitive circuits. In the next subsection we give examples of such circuits that are highly repetitive for a reasonable range of parties.

For the remainder of this paper, we will use w denote the maximum width of the circuit C, w_m to denote the width of the mth layer and $w_{m,j}$ to denote the width of $\mathsf{block}_{m,j}$.

3.3.3 Examples of Highly Repetitive Circuits

We highlight 3 functionalities with circuit representations that are part of the highly repetitive circuit class. First, we describe machine learning circuits, focusing on training algorithms that leverage gradient decent. Then, we discuss cryptographic hash functions like SHA256 and block ciphers like AES.

Machine Learning. Machine learning algorithms extract trends from large datasets to facilitate

accurate prediction in new, unknown circumstances. Training can been viewed as an optimization problem, in which the model attempts to find internal parameters that minimizes the error between its predictions and ground truth. A common family of algorithms for minimizing this error is called "gradient decent." Starting with random internal parameters, the algorithm iteratively reduces the error by making a small, greedy changes. When run without privacy, the algorithm terminates when it converges (*i.e.* the marginal decrease in error is zero). However, because MPC computation must be data oblivious, the number of iterations must be selected *before execution* and must cover the worst case scenario. Different versions of this algorithm are used to train simple models, like linear regression, or more complex and powerful models, like neural networks. For a more complete description of gradient decent training algorithms, and their adaptation to MPC, see [MR18].

The exact number of gates in the circuit representation of privacy-preserving model training is difficult to calculate from prior work. In one of the few concrete estimates, Gascón et al. [GSB+16] realize coordinate gradient decent training algorithms with approximately 10^{11} gates. As noted in [MZ17], the storage requirement for this circuit would be 3000GB. Subsequent work stopped estimating gate counts altogether, instead building a library of sub-circuits that can be loaded as needed. As the amount of data used to train models continues to grow, circuits sizes will continue to increase. While we are not able to accurately estimate the number of gates for this kind of circuit, we can still establish that their structure is highly repetitive. For instance, the gradient decent algorithm consists of nothing but iterations of the same functionality. In the implementation of Mohassel et al. [MR18], the default configuration for training is 10000 iterations, clearly enough repeated depth to accommodate massive number of players. Indeed, in the worst case the depth of a gradient decent algorithm must be linear in the input size. This is because gradient decent usually uses a *batching* technique, in which only a subset of the data is used for any given iteration. However, as all the algorithm wants is to accommodate as much new data as possible, the number of batches should be linear in the input size.

The width of gradient decent training algorithms is usually roughly proportional to the dimension of the dataset. For most interesting applications of machine learning, high dimensional data is

Table 3.1: Size of the highly repetitive circuits we consider in this work.

Circuit	Gates ($\mathbb{F}_2$)	Iterative Loops	Gates per Loop	Percent Repeated Structure
SHA256 (1 Block)	119591	64	1437	77%
AES128 (1 Block)	7458	10	656	88%
Gradient Descent	—	≥ 10000	—	$\sim 100\%$

normal. If a particular application does not have high enough dimension to allow massive number of parties to participate in the protocol, we note that parallelism can be leveraged. Specifically, gradient decent training algorithms usually use a *random restart* strategy to avoid getting trapped at local minima. These independent runs of the algorithm can be run in parallel, making the circuit quite wide. Some final logic may be added at the end to select the output from the iterations that produced optimal internal parameters.

Cryptographic Hash Functions. All currently deployed cryptographic hash functions rely on iterating over a round function. This round function typically has a diffusion property such that, after many invocations, it is widely considered impossible to invert. Importantly for our purposes, each iteration of the round function is (typically) *structurally identical*. Moreover, the vast majority of the gates in the circuit representation of a hash function are contained within the iterations of the round function. As a concrete study of such a cryptographic hash function, we consider SHA256 [NIS02]. SHA256 is one of the most widely deployed hash functions; given its common use in applications like Bitcoin [Nak08] and ECDSA [GFD09], SHA256 is an important building block of MPC applications. SHA256 contains 64 rounds of its inner function, with other versions that use larger block size containing 80 rounds.

To measure the proportion of the SHA256 circuit that is contained within the iterated round function, we implement a Frigate [MGC+16] compatible SHA256 description for hashing a single block of input. While our protocol is intended for arithmetic circuits, but there are no well tested arithmetic circuit compilers and our protocol can be adapted to binary field. As can be seen in Table 3.1, 77% of all the gates in the compiled SHA256 are repeated structure, that structure repeating at least 64 times.

We note that these results were for hashing only a single block of input. When hashing a single block of input, there are gates to handle initialization and output, comprising the remaining 23% of gates. However, it is unlikely that an MPC with hundreds or thousands of players will compute only a single block of SHA256; it is more plausible that each participating player will contribute additional data, for $O(n)$ total blocks. These additional blocks of input do not contain the overhead, so all the additional gates will comprise repeated structure. For instance, if there are as few as 10 blocks of input, the circuit is already 97% repeated structure.

If we consider the case where the number of blocks of input is proportional to the number of player, all that remains to argue is that the width of the circuit is sufficient that each gate block is sufficiently large. As mentioned, there are no good arithmetic compilers available, so it is difficult to argue about the width of the arithmetic circuit computing the functionality SHA256. We note that the width of a block is 512 bits. If width is proportional to this, it is very plausible to say hundreds of players could compute this functionality. However, when computing over a larger field, there may not be enough gates in each layer. As such, we note that there are many common applications which require many *parallel* iterations of hash functions. For instance, if players wish to compute a Merkle tree over their inputs, the resulting circuit will naturally satisfy our requirements.

Block Ciphers. Modern block ciphers, similar to cryptographic functions, are iterative by nature. Advanced Encryption Standard, the block cipher on which we focus, uses either 10, 12, or 14 iterations of its round function, depending on the key length used. The round function is comprised of a substitution step, a shifting step, a mixing step, with all but one iteration containing all of these steps. Again, this repeated structure allows the pre-processing phase of our protocol to be run very efficiently. Performing a similar analysis as with SHA256, we identified that 88% of the gates in AES128 are part of this repeated structure when encrypting a single block of input. Just as with hash functions, more blocks of input lead to increased percentage repeated structure. With 10 blocks of input, 98% of the gates are repeated structure.

As with hash functions, we note that width may be a concern for applying our protocol. However,

computing many parallel encryptions is also a common task. For instance, if players wish to encrypt or decrypt a disk image, encrypting under multiple keys is common. These different sectors can be evaluated in parallel, giving sufficient structure.

3.3.4 Protocol Switching for Circuits with Partially Repeated Structure

Hash functions and symmetric key cryptography are not comprised of 100% repeated structure. When structure is not repeated, the batched randomness generation step cannot be run efficiently. In the worst case, if a particular piece of structure is only present once in the circuit, $O(n^2)$ messages will be used to generate only a single packet secret share of size $O(n)$. If $0 \leq p \leq 1$ is the fraction of the circuit that is repeated, our protocol has efficiency $O(p|C| + (1-p)n|C|)$.

We note that our protocol has worse constants than [CGH+18] and [FL19] when run on the non-repeated portion of the circuit. Specifically, our protocol requires communication for all gates, rather than just multiplication gates. As we are trying to push the constants as low as possible, it would be ideal to run the most efficient known protocols for the portions of the circuit that are linear in the number of players. To do this, we note that our protocol can support mid-evaluation protocol switching.

Recall our simple non-interactive technique to transform normal secret shares into packed secret shares, presented in Section 3.4.3. This technique can be used in the middle of protocol execution to switch between a traditional, efficient, $O(n|C|)$ protocol and our protocol. Once the portion of the circuit without repeated structure is computed using another efficient protocol, the players can pause to properly structure their secret shares and non-interactively pack them. The players can then evaluate the circuit using our protocol. If another patch of non-repeated structure is encountered, the players can use the leader to reconstruct and re-share normal shares as necessary. Importantly, because all of these protocols are linear, it is still possible to use the malicious security compiler of [CGH+18].

3.4 Input Sharing Phase

In this section, we present the sub-protocols/functionalities that will be used for secret sharing inputs in our main protocols. We begin by describing the functionality for generating (regular) shares for random values in Section 3.4.1. Then in Section 3.4.2, we show how the parties can use the previous functionality for computing (regular) shares of their inputs. Then in Section 3.4.3, we describe a non-interactive transformation that allows a set of parties holding shares corresponding to ℓ secrets, to compute a single packed secret sharing of the vector containing those ℓ secrets. Finally, in Section 3.4.4, we show how the above protocols can be combined to enable parties to obtain packed secret sharings of their inputs.

3.4.1 Generating Shares of Random Values

In this section, we describe a protocol π_{rand} for generating (regular) shares of a batch of random and independently chosen values (this is identical to the protocol proposed in [DN07]). In our main protocol, π_{rand} will help us robustly share inputs.

This protocol either outputs honestly computed (regular) shares of random values or it outputs $\perp$. It makes use of the regular Shamir's secret sharing scheme along with an $n \times n$ hyper-invertible matrix. First, each party samples a random value and (regular) secret shares it among the other parties. The parties compute n linear combinations of these shares using the Vandermonde matrix. The parties then open t sets of resulting shares to all the parties, who locally verify the correctness of these shares. If all n parties are happy with their checks, the remaining $n-t$ shares are output by the protocol. If the check succeeds, then the hyper-invertability property of guarantees that the remaining $n-t$ shares are random and honestly generated. We now proceed to formally define the f_{rand} functionality and then describe a protocol that securely computes $n-t$ instantiations of f_{rand} with abort. As discussed earlier, here we will work with a slightly modified sharing algorithm of the Shamir's secret sharing scheme (see Section 2.2.1 for details).

The ideal functionality realized by this protocol is described in Figure 3.3. Since the adversary

can choose its own shares in the protocol, similar to Chida et. all [CGH+18], we let adversary send shares of the corrupted parties to the ideal functionality.

The functionality $f_{\mathsf{rand}}(\{P_1, \ldots, P_n\})$

The n-party functionality f_{rand}, running with parties $\{P_1, \ldots, P_n\}$ and the ideal adversary Sim proceeds as follows:

- The ideal simulator Sim sends u_i for each corrupt party $i \in \mathcal{A}$.
- The functionality f_{rand} chooses a random value $r \in \mathbb{F}$, sets $[r]_{\mathcal{A}} = \{u_i\}_{i \in \mathcal{A}}$. It runs $\mathsf{share}(r, \mathcal{A}, [r]_{\mathcal{A}}, t + \ell)$ to receive a share r_i for each party P_i.
- It hands each honest party P_j its share r_j.

Figure 3.3: Random share generation functionality

We now describe the protocol π_{rand} that securely realizes this functionality f_{rand} (Figure 3.3). The protocol proceeds as follows:

Auxiliary Inputs Hyper-invertible matrix $\mathbf{H}_{n,n}$

Inputs: The parties do not have any inputs.

Protocol π_{rand}**:** The parties proceed as follows:

- Each party $\{P_i\}$ (for $i \in [n]$) chooses a random element $u_i \in \mathbb{F}$. It runs $\mathsf{share}(u_i, t+\ell)$ to receive shares $[u_i]$. For each $j \in [n]$, it party P_j, its share in $[u_j]$.
- Given shares $([u_1], \ldots [u_n])$, the parties compute

$$([r_1], \ldots, [r_n]) = \mathbf{H}^T_{n,n} \cdot ([u_1], \ldots, [u_n])$$

- Each party sends its shares in $[r_{n-t+1}], \ldots, [r_n]$ to all other parties. The parties locally run $\mathsf{open}([r_{n-t+1}]), \ldots, \mathsf{open}([r_n])$ to check if all the shares lie on the same degree $t + \ell$ polynomial and moreover that the polynomial is of the form $r_{t+\ell} + q(z) \prod_{j=1}^{\ell}(z - e_j)$, where $q(z)$ is a degree t polynomial. If this check succeeds, then the parties send "pass" to all other parties, else they send "fail".

– If all n parties output pass, then the parties output their shares in $[r_1], \ldots, [r_{n-t}]$, else they output $\perp$ and halt.

Output: The parties output $[r_1], \ldots [r_{n-t}]$ or $\perp$.

Lemma 1. *This protocol securely computes $n-t$ instantiations of f_{rand} with abort in the presence of malicious adversaries who controls t parties.*

The proof of this Lemma follows from [BTH08], hence we omit it here.

3.4.2 Secret Sharing of Inputs

In this section, we describe a well known protocol π_{input} for generating honest shares of each parties' inputs. We borrow much of the language from Chida et. al in [CGH+18] for this description. This sub-protocol will be used in our protocol to give robust sharings of inputs. Note that because we operate on packed secret shares, this protocol alone is not sufficient to prepare inputs for evaluation. We describe a non-interactive way of transforming these robust shares into (robust) packed secret shares in the next section.

For each input x_i belonging to party P_i, the parties invoke f_{rand} to generate a random sharing $[r_i]$. They open the value of r to the designated owner P_i of x_i. P_i reconstructs r_i, computes $x_i - r_i$ and sends $x_i - r_i$ to all the parties. Each party then adds this value to its respective share of r_i. Since f_{rand} ensures that $[r]$ is an honest sharing of r, this in turn ensures the sharing of x_i is also honest. The ideal functionality realized by this protocol is described in Figure 3.4.

We now describe the protocol π_{input} that securely realizes this functionality f_{input} (Figure 3.4). The protocol proceeds as follows:

Inputs: Let $x_1, \ldots, x_M \in \mathbb{F}$ be the series of inputs, each x_i is held by some party P_j.

Protocol π_{input}: The parties proceed as follows:

– The parties $\{P_1, \ldots, P_n\}$ invoke f_{rand} M times to obtain sharings $[r_1], \ldots, [r_M]$.

– For each $i \in [M]$, all the parties send their shares in $[r_i]$ to party P_j, who owns the input. Party

The functionality $f_{\mathsf{input}}(\mathcal{P} := \{P_1, \ldots, P_n\},)$

The functionality f_{input}, running with parties $\{P_1, \ldots, P_n\}$ and the ideal adversary Sim proceeds as follows:

– It receives inputs $x_1, \ldots, x_M \in \mathbb{F}$ from the respective parties.

– For every $i \in [M]$, f_{input} also receives from Sim the shares $[x_i]_{\mathcal{A}}$ of the corrupted parties for the ith input.

– For every $i \in [M]$, f_{input} computes all shares $(x_{i,1}, \ldots, x_{i,n}) = \mathsf{share}(x_i, \mathcal{A}, [x_i]_{\mathcal{A}}, t+\ell)$.

– For every $j \in [n]$, f_{input} sends P_j its output shares $(x_{1,j}, \ldots, x_{M,j})$.

Figure 3.4: Secret sharing of inputs functionality

P_j runs $\mathsf{open}([r_i])$. If it receives $\perp$, then it sends $\perp$ to all parties, outputs `abort` and halts.

– For each $i \in [M]$, party P_j (who owns input x_i) sends $v_i = x_i - r_i$ to all other parties.

– All parties send $\overrightarrow{v} = (v_2, \ldots, v_M)$ to all other parties. If any party receives a different vector to its own, then it outputs $\perp$ and halts.

– For each $i \in [M]$, the parties compute $[x_i] = [r_i] + v_i$.

Output: The parties output $[x_1], \ldots, [x_M]$

Lemma 2. *This protocol securely computes f_{input} with abort in the f_{rand}-hybrid model in the presence of a malicious adversary who controls at most t parties.*

Proof. Let $\mathcal{A}$ be the real adversary. We construct a simulator Sim as follows. Sim receives $[r_i]_{\mathcal{A}}$ for each $i \in [M]$ that $\mathcal{A}$, sends to f_{rand} in the protocol. For each $i \in [M]$, it samples random $r_i \in \mathbb{F}$ and computes $[r_i] \leftarrow \mathsf{share}(r_i, \mathcal{A}, [r_i]_{\mathcal{A}}, t+\ell)$. Sim then simulates the honest parties in all reconstruct executions. If an honest party P_j receives $\perp$ in the reconstruction, then Sim simulates it sending $\perp$ to all parties. Sim simulates the remainder of the execution, obtaining all v_i values from $\mathcal{A}$ associated with the corrupted parties' inputs, and sending random v_j values for inputs associated with honest parties. For every i for which the ith input is that of a corrupted party P_i, Sim sends $x_i = v_i + r_i$ to the ideal functionality f_{input}. For every $i \in [n]$, Sim defines the corrupted parties' shares $[x_i]_{\mathcal{A}}$ to be

$[r_i + v_i]_{\mathcal{A}}$. Then Sim sends $[x_i]_{\mathcal{A}}, \ldots, [x_n]_{\mathcal{A}}$ to the ideal functionality f_{input}. For every honest party, if it aborted in the simulation, then Sim sends `abort` to the ideal functionality f_{input}, else, it sends continue. Finally Sim outputs whatever $\mathcal{A}$ outputs. While indistinguishability of the honest parties output follows trivially, indistinguishability of the corrupt parties' view in the real and ideal worlds follows from the fact that the adversary only gets to see t shares of the honest parties' inputs. From the privacy property of secret sharing, we know that t shares are not sufficient for reconstructing shares corresponding to a $t + \ell$ degree polynomial.

□

3.4.3 A Non-Interactive Protocol for Packing Regular Secret Shares

We now describe a novel, non-interactive transformation that allows a set of parties holding shares corresponding to ℓ secrets $[s_1], \ldots, [s_\ell]$ to compute a single packed secret sharing of the vector $\mathbf{v} = (s_1, \ldots, s_\ell)$. This protocol makes a non-black-box use of Shamir secret sharing to accomplish this packing without interaction. As discussed in the technical overview, to achieve efficiency, our protocol computes over packed shares. But, if each player follows the naïve strategy of just packing all their own inputs into a single vector, the values may not be properly aligned for computation. This non-interactive functionality lets players simply share their inputs using f_{input}, which is a simple input sharing functionality based on Shamir secret sharing (see Section 3.4.2), and then locally pack the values in a way that guarantees alignment.

Let $p_1, \ldots, p_\ell$ be the degree $t + \ell$ polynomials that were used for secret sharing secrets $s_1 \ldots, s_\ell$ respectively such that each $p_i(z)$ (for $i \in [\ell]$) is of the form $s_i + q_i(z) \prod_{j=1}^{\ell}(z - e_j)$, where q_i is a degree t polynomial. Then each party P_j (for $j \in [n]$) holds shares $p_1(\alpha_j), \ldots, p_\ell(\alpha_j)$.

Given these shares, each party P_j computes a packed secret share of the vector $(s_1, \ldots, s_\ell)$ as follows:

$$\mathcal{F}_{\mathsf{SS-to-PSS}}(\{p_i(\alpha_j)\}_{i \in [\ell]}) = \sum_{i=1}^{\ell} p_i(\alpha_j) L_i(\alpha_j) = p(\alpha_j)$$

where $L_i(\alpha_j) = \prod_{j=1, j \neq i}^{\ell} \frac{(\alpha_i - e_j)}{(e_i - e_j)}$ is the Lagrange interpolation constant and p corresponds to a new

degree $D = t + 2\ell - 1$ polynomial for the packed secret sharing of vector $\mathbf{v} = (s_1, \ldots, s_\ell)$.

Lemma 3. *For each $i \in [\ell]$, let $s_a \in \mathbb{F}$ be secret shared using a degree $t + \ell$ polynomial p_i of the form $s_i + q_i(z) \prod_{j=1}^{\ell}(z - e_j)$, where q_i is a degree t polynomial and $e_1, \ldots, e_\ell$ are some pre-determined field elements. Then for each $j \in [n]$, $\mathcal{F}_{\mathsf{SS-to-PSS}}(\{p_i(\alpha_j)\}_{i \in [\ell]})$ outputs the jth share corresponding to a valid packed secret sharing of the vector $\mathbf{v} = (s_1, \ldots, s_\ell)$, w.r.t. a degree-$D = t + 2\ell - 1$ polynomial.*

Proof. For each $i \in [\ell]$, let $p_i(z)$ be the polynomial used for secret sharing the secret s_i. We know that $p_i(z)$ is of the form

$$p_i(z) = s_i + q_i(z) \prod_{j=1}^{\ell}(z - e_j),$$

where q_i is a degree t polynomial. Let $p'_i(z) = q_i(z) \prod_{j=1}^{\ell}(z - e_j)$ and let $p(z)$ be the new polynomial corresponding to the packed secret sharing. From the description of $\mathcal{F}_{\mathsf{SS-to-PSS}}$, it follows that:

$$\begin{aligned} p(z) =& \sum_{i=1}^{\ell} p_i(z) L_i(z) = \sum_{i=1}^{\ell} p'_i(z) L_i(z) + s_i L_i(z) \\ =& \sum_{i=1}^{\ell} p'_i(z) \prod_{j=1, j \neq i}^{\ell} \frac{(z - e_j)}{(e_i - e_j)} + \sum_{i=1}^{\ell} s_i L_i(z) \\ =& \sum_{i=1}^{\ell} q_i(z) \prod_{j=1, j \neq i}^{\ell} \frac{(z - e_j)}{(e_i - e_j)} \prod_{j=1}^{\ell}(z - e_j) + \sum_{i=1}^{\ell} s_i L_i(z) \end{aligned}$$

$$\begin{aligned} \text{Let } q'_i(z) =& q_i(z) \prod_{j=1, j \neq i}^{\ell} \frac{(z - e_j)}{(e_i - e_j)} \\ p(z) =& (q'_1(z) + \ldots + q'_\ell(z)) \prod_{j=1}^{\ell}(z - e_j) + \sum_{i=1}^{\ell} s_i L_i(z) \\ =& q(z) \prod_{j=1}^{\ell}(z - e_j) + \sum_{i=1}^{\ell} s_i L_i(z) \end{aligned}$$

where $q(z) = q'_1(z) + \ldots + q'_\ell(z)$ is a degree $t + \ell - 1$ polynomial and hence $p(z)$ is a degree $D = t + 2\ell - 1$ polynomial. It is now easy to see that for each $i \in [\ell]$, $p(e_i) = s_i$. Hence $\mathcal{F}_{\mathsf{SS-to-PSS}}$ computes a valid packed secret sharing of the vector $\mathbf{v} = (s_1, \ldots, s_\ell)$. □

3.4.4 Packed Secret Sharing of Inputs

We now arrive at a subprotocol that will be invoked directly in our protocol execution. This functionality takes in the individual inputs of the players and outputs a packed secret sharing of these inputs. Using the circuit information, players can run $\mathsf{WireConfiguration}(\mathsf{block}_{0,j}, \mathsf{block}_{1,j})$ for each $j \in [\sigma]$ to determine the alignment of vectors required to compute the first layer of the circuit. Because each $\mathsf{block}_{1,j}$ in the circuit contains $w_{1,j}/\ell$ gates, the protocol outputs $2w_1/\ell = \sum_{j\in[\sigma]} w_{1,j}$ properly aligned packed secret shares, each containing ℓ values. This functionality makes use of our non-interactive packing protocol described in Section 3.4.3. A formal description of the ideal functionality for this subprotocol appears in Figure 3.5.

The functionality $f_{\mathsf{pack-input}}(\mathcal{P} := \{P_1, \ldots, P_n\},)$

The functionality $f_{\mathsf{pack-input}}$, running with parties $\{P_1, \ldots, P_n\}$ and the ideal adversary Sim proceeds as follows:

- It receives inputs $x_1, \ldots, x_M \in \mathbb{F}$ from the respective parties and the layers $\mathsf{layer}_0, \mathsf{layer}_1$ from all parties.
- It computes $\{\mathsf{block}_{0,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(0, \mathsf{layer}_0)$ and $\{\mathsf{block}_{1,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(1, \mathsf{layer}_1)$.
- For each $j \in [\sigma]$, it computes $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{1,j}, \mathsf{block}_{0,j})$.
- For each $j \in [\sigma]$ and $q \in [w_{1,j}/\ell]$,
  - Set $\mathbf{x}^{j,q} = (x_{\mathsf{LeftInputs}_j[i]})_{i\in\{(q-1)\ell+1,\ldots,q\ell\}}$ and $\mathbf{y}^{j,q} = (x_{\mathsf{RightInputs}_j[i]})_{i\in\{(q-1)\ell+1,\ldots,q\ell\}}$.
  - Receives from Sim, the shares $[\mathbf{x}^{j,q}]_{\mathcal{A}}, [\mathbf{y}^{j,q}]_{\mathcal{A}}$ of the corrupted parties for the input vectors $\mathbf{x}_{j,q}, \mathbf{y}_{j,q}$.
  - It computes shares $\mathbf{x}^{j,q} \leftarrow \mathsf{pshare}(\mathbf{x}^{j,q}, \mathcal{A}, [\mathbf{x}^{j,q}]_{\mathcal{A}}, D)$ and $\mathbf{y}^{j,q} \leftarrow \mathsf{pshare}(\mathbf{y}^{j,q}, \mathcal{A}, [\mathbf{y}^{j,q}]_{\mathcal{A}}, D)$ and sends them to the parties.

Figure 3.5: Packed Secret sharing of all inputs functionality

We now describe the protocol $\pi_{\mathsf{pack-input}}$ that securely realizes this functionality $f_{\mathsf{pack-input}}$ (Figure 3.5). The protocol proceeds as follows:

Inputs: Let $x_1, \ldots, x_M \in \mathbb{F}$ be the series of inputs, each x_i is held by some party P_j.

Protocol $\pi_{\mathsf{pack-input}}$**:** The parties proceed as follows:

1. For each $i \in [M]$, the parties invoke f_{input} on x_i to obtain regular shares $[x_i]$.

2. The parties locally compute $\{\mathsf{block}_{0,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(0, \mathsf{layer}_0)$ and $\{\mathsf{block}_{1,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(1, \mathsf{layer}_1)$.

3. For each $j \in [\sigma]$, the parties compute

$$\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{1,j}, \mathsf{block}_{0,j}).$$

4. For each $j \in [\sigma]$ and $q \in [w_{1,j}/\ell]$, they compute packed secret sharing of vectors $\mathbf{x}^{j,q}$ and $\mathbf{y}^{j,q}$ as follows:

$$[\mathbf{x}^{j,q}] = \mathbb{F}_{\mathsf{SS-to-PSS}}(([x_{\mathsf{LeftInputs}_j[i]}])_{i\in\{(q-1)\ell+1,\ldots,q\ell\}})$$

$$[\mathbf{y}^{j,q}] = \mathbb{F}_{\mathsf{SS-to-PSS}}([y_{\mathsf{RightInputs}_j[i]}])_{i\in\{(q-1)\ell+1,\ldots,q\ell\}}$$

Output: Each party outputs its shares in $\left\{[\mathbf{x}^{j,q}], [\mathbf{y}^{j,q}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$.

Lemma 4. *This protocol securely computes $f_{\mathsf{pack-input}}$ with abort in the f_{input}-hybrid model in the presence of malicious adversaries who control at most t parties.*

Proof. The proof of this lemma follows trivially from the correctness of the non-interative transformation from regular secret sharing to packed secret sharing (Lemma 3). □

3.5 Circuit Evaluation Phase

In this section, we present the sub-protocols that will be used in the online evaluation of the circuit. In Section 3.5.1, we present our randomness generation sub-protocol that outputs packed shares of correlated random values, where the correlation is dictated by the configuration of the circuit. Then in Section 3.5.2, we present our main circuit evaluation subprotocol, that takes the random shares generated by the previous protocol and packed shares of input vectors output by the subprotocol from Section 3.4.4 to evaluate the circuit layer-wise.

3.5.1 Generating Correlated Random Packed Sharings

We now turn to the randomness generation protocol for our main construction. Recall from the technical overview that the packed secret sharings of random values must be generated according to the circuit structure. More specifically, the unmasking values (degree D shares) for some $\mathsf{block}_{m+1,j}$ must be aligned according to the output of $\mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$.

Before describing the protocol, we quickly note the number of shares that it generates, as it is somewhat non-standard. Let $w_{m,j}$ be the number of gates in $\mathsf{block}_{m,j}$ and $w_{m+1,j}$ be the number of gates in $\mathsf{block}_{m+1,j}$. As noted in the technical overview, our protocol treats each gate as though it performs *all* operations (relay, addition and multiplication). This lets the players evaluate different operations on each value over packed secret shares. Each of these operations must be masked with different randomness to ensure privacy. As such, the protocol generates $3w_{m,j}/\ell$ shares of uniform random vectors. To facilitate unmasking after the leader has run the realignment procedure, the protocol must generate shares of vectors with values selected from these $3w_{m,j}/\ell$ random vectors. This selection is governed by $\mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$. Since there are $w_{m+1,j}$ gates in $\mathsf{block}_{m+1,j}$, the functionality will output $2w_{m+1,j}/\ell$ of these unmasking shares (with degree D). In total, these are $(3w_{m,j} + 2w_{m+1,j})/\ell$ packed secret sharings.

To summarize, this protocol has the following main steps:

1. The parties generate $3w_{m,j}/\ell$ uniform random vectors, corresponding to the values that will be used to mask the outputs of $\mathsf{block}_{m,j}$.

2. Parties compute $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$ to determine the required alignment of the correlated random vectors.

3. Parties use $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j$ and the gate information of $\mathsf{block}_{m,j}$ to select the appropriate values from step 1 for unmasking shares. This results in $2w_{m+1,j}/\ell$ vectors

4. Parties share these $(3w_{m,j} + 2w_{m+1,j})/\ell$ vectors using packed secret sharing and deal the resulting shares to all parties.

5. Parties use the Vandermonde matrix $\mathbf{V}_{n,(n-t)}$ to compute linear combinations of the shares they have received, and output the result.

We now give a formal description of this subprotocol $\pi_{\mathsf{corr-rand}}$.

Auxiliary Inputs Vandermonde matrix $\mathbf{V}_{n,(n-t)} \in \mathbb{F}^{n\times(n-t)}$.

Inputs: All parties get a configuration block pair $(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$ as input.

Protocol $\pi_{\mathsf{corr-rand}}$**:** The parties proceed as follows:

– Each party P_i (for $i \in [n]$) chooses $3w_{m,j}/\ell$ random vectors $(\{\mathbf{s}_i^{q,\mathsf{mult}}, \mathbf{s}_i^{q,\mathsf{add}}, \mathbf{s}_i^{q,\mathsf{relay}}\}_{q\in[3w_{m,j}/\ell]}) \in \mathbb{F}^{\ell\times w_{m,j}/\ell}$ of length ℓ each.

– The parties compute $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$.

– For each $q \in [w_{m+1,j}/\ell]$ and for each $k \in [\ell]$, let $e_{\mathsf{left}} = \mathsf{LeftInputs}_j[(q-1)\ell + i]$ and $e_{\mathsf{right}} = \mathsf{RightInputs}_j[(q-1)\ell + i]$ and the parties set:

$$\mathbf{s}_i^{q,\mathsf{left}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{left}}/\ell\rfloor,\mathsf{GateType}_k}[e_{\mathsf{left}} - \lfloor e_{\mathsf{left}}/\ell\rfloor]$$

$$\mathbf{s}_i^{q,\mathsf{right}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{right}}/\ell\rfloor,\mathsf{GateType}_k}[e_{\mathsf{right}} - \lfloor e_{\mathsf{right}}/\ell\rfloor]$$

where $\mathsf{GateType}_k = \mathsf{mult}$ if gate k in block m, j is a multiplication gate, else if it is an addition gate then $\mathsf{GateType}_k = \mathsf{add}$ and for relay gates, $\mathsf{GateType}_k = \mathsf{relay}$.

– For each $q \in [w_{m,j}/\ell]$ and $\mathsf{GateType} \in \{\mathsf{add}, \mathsf{relay}, \mathsf{mult}\}$, the parties compute

$$\langle \mathbf{s}_i^{q,\mathsf{GateType}} \rangle = \mathsf{pshare}(\mathbf{s}_i^{q,\mathsf{GateType}}, n-1)$$

and for each $q \in [w_{m+1,j}/\ell]$, the parties compute

$$[\mathbf{s}_i^{q,\mathsf{left}}] = \mathsf{pshare}(\mathbf{s}_i^{q,\mathsf{left}}, D), \quad [\mathbf{s}_i^{q,\mathsf{right}}] = \mathsf{pshare}(\mathbf{s}_i^{q,\mathsf{right}}, D)$$

and sends the respective shares to each party.

– Given these shares, for each $q \in [w_{m,j}/\ell]$ and GateType $\in \{\mathsf{add}, \mathsf{relay}, \mathsf{mult}\}$, the parties compute the following:

$$(\langle \mathbf{r}_1^{q,\mathsf{GateType}} \rangle, \ldots, \langle \mathbf{r}_{n-t}^{q,\mathsf{GateType}} \rangle) = \mathbf{V}_{n,(n-t)} \cdot (\langle \mathbf{s}_1^{q,\mathsf{GateType}} \rangle, \ldots, \langle \mathbf{s}_n^{q,\mathsf{GateType}} \rangle)$$

and for each $q \in [w_{m+1,j}/\ell]$, they compute

$$([\mathbf{r}_1^{q,\mathsf{left}}], \ldots, [\mathbf{r}_{n-t}^{q,\mathsf{left}}]) = \mathbf{V}_{n,(n-t)} \cdot ([\mathbf{s}_1^{q,\mathsf{left}}], \ldots, [\mathbf{s}_n^{q,\mathsf{left}}])$$

$$([\mathbf{r}_1^{q,\mathsf{right}}], \ldots, [\mathbf{r}_{n-t}^{q,\mathsf{right}}]) = \mathbf{V}_{n,(n-t)} \cdot ([\mathbf{s}_1^{q,\mathsf{right}}], \ldots, [\mathbf{s}_n^{q,\mathsf{right}}])$$

– The parties output their shares

$$\{\{[\mathbf{r}_i^{q,\mathsf{left}}], [\mathbf{r}_i^{q,\mathsf{right}}]\}_{q \in [w_{m+1,j}/\ell]}, \{\langle \mathbf{r}_i^{q,\mathsf{mult}} \rangle, \langle \mathbf{r}_i^{q,\mathsf{add}} \rangle, \langle \mathbf{r}_i^{q,\mathsf{relay}} \rangle\}_{q \in [w_{m,j}/\ell]}\}_{i \in [n-t]}.$$

3.5.2 Secure Layer-Wise Circuit Evaluation

This sub-protocol evaluates the circuit in a layer-wise fashion, i.e., it evaluate all gates in a given layer simultaneously. It takes properly aligned input vectors $\left\{[\mathbf{x}_1^{j,q}], [\mathbf{y}_1^{j,q}]\right\}_{j \in [\sigma], q \in [w_{1,j}/\ell]}$ held by a set of parties, and computes packed shares $[\mathbf{z}^{j,q,\mathsf{left}}]$ and $[\mathbf{z}^{j,q,\mathsf{right}}]$, for each $m \in [d+1]$, $j \in [\sigma]$ and $q \in [w_{m+1,j}/\ell]$. We note that for notational convenience, this sub-protocol takes as input $\left\{[\mathbf{x}_1^{j,q}], [\mathbf{y}_1^{j,q}], [\mathbf{x}_2^{j,q}], [\mathbf{y}_2^{j,q}]\right\}_{j \in [\sigma], q \in [w_{m,j}/\ell]}$ instead of just $\left\{[\mathbf{x}_1^{j,q}], [\mathbf{y}_1^{j,q}]\right\}_{j \in [\sigma], q \in [w_{m,j}/\ell]}$. This is because in our maliciously secure protocol, we invoke this sub-protocol for evaluating the circuit on actual inputs as well as on randomized inputs. When computing on actual inputs, we set $\mathbf{x}_1^{j,q} = \mathbf{x}_2^{j,q}$ and $\mathbf{y}_1^{j,q} = \mathbf{y}_2^{j,q}$ and when computing on randomized inputs, we set $\mathbf{x}_2^{j,q} = \mathbf{r}\mathbf{x}_1^{j,q}$ and $\mathbf{y}_2^{j,q} = \mathbf{r}\mathbf{y}_1^{j,q}$. A detailed description of this sub-protocol appears in Figure 3.6.

The protocol $\pi_{\mathsf{eval}}(\{P_1, \ldots, P_n\})$

Input: The parties $\{P_i\}_{i\in[n]}$ hold packed secret sharings $\left\{[\mathbf{x}_1^{j,q}], [\mathbf{y}_1^{j,q}], [\mathbf{x}_2^{j,q}], [\mathbf{y}_2^{j,q}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$ and for each $m \in [d]$, they hold configuration of layers layer_m and layer_{m+1}. For each $m \in [d]$, let $\{\mathsf{block}_{m,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m, \mathsf{layer_m})$ and $\{\mathsf{block}_{m+1,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m+1, \mathsf{layer_{m+1}})$. Let $\mathsf{Unique} \subseteq \{(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})\}_{d\in[m],j\in[\sigma]}$ be such that for every pair $(\mathsf{block}_{a+1}, \mathsf{block}_a), (\mathsf{block}_{b+1,b}, \mathsf{block}_b) \in \mathsf{Unique}$, it holds that $\mathsf{WireConfiguration}(\mathsf{block}_{a+1}, \mathsf{block}_a) \neq \mathsf{WireConfiguration}(\mathsf{block}_{b+1}, \mathsf{block}_b)$.

Protocol: The parties proceed as follows:

- *Generating Correlated Randomness:* For each $(\mathsf{block}_{a+1}, \mathsf{block}_a) \in \mathsf{Unique}$, the parties run $\pi_{\mathsf{corr-rand}}$ to obtain packed secret shares $\{\{[\mathbf{r}_i^{q,\mathsf{left}}], [\mathbf{r}_i^{q,\mathsf{right}}]\}_{q\in[w_{a+1}/\ell]}, \{\langle\mathbf{r}_i^{q,\mathsf{mult}}\rangle, \langle\mathbf{r}_i^{q,\mathsf{add}}\rangle, \langle\mathbf{r}_i^{q,\mathsf{relay}}\rangle\}_{q\in[w_a/\ell]}\}_{i\in[n-t]}$, where w_a and w_{a+1} are the lengths of blocks block_a and block_{a+1} respectively. The parties then assign these shares to different blocks in the circuit based on the configuration of each block. In other words, we assume that at the end of this step for each $m \in [d], j \in [\sigma]$, the parties have the following shares:

$$\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}], [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]\}_{j,q\in[w_{m+1,j}/\ell]}, \{\langle\mathbf{r}_m^{j,q,\mathsf{mult}}\rangle, \langle\mathbf{r}_m^{j,q,\mathsf{add}}\rangle, \langle\mathbf{r}_m^{j,q,\mathsf{relay}}\rangle\}_{q\in[w_{m,j}/\ell]}$$

- *Layer-wise Evaluation:* Circuit evaluation proceeds layer-wise, where for each $m \in [d], j \in [\sigma]$, the parties proceed as follows:
  - For each $q \in [w_{m,j}/\ell]$, the parties locally compute the following:
    $\langle\mathbf{x}_1^{j,q} \cdot \mathbf{y}_2^{j,q} + \mathbf{r}_m^{j,q,\mathsf{mult}}\rangle = [\mathbf{x}_1^{j,q}] \cdot [\mathbf{y}_2^{j,q}] + \langle\mathbf{r}_m^{j,q,\mathsf{mult}}\rangle$
    $\langle\mathbf{x}_1^{j,q} + \mathbf{y}_1^{j,q} + \mathbf{r}_m^{j,q,\mathsf{add}}\rangle = [\mathbf{x}_1^{j,q}] + [\mathbf{y}_1^{j,q}] + \langle\mathbf{r}_m^{j,q,\mathsf{add}}\rangle$
    $\langle\mathbf{x}_1^{j,q} + \mathbf{r}_m^{j,q,\mathsf{relay}}\rangle = [\mathbf{x}_1^{j,q}] + \langle\mathbf{r}_m^{j,q,\mathsf{relay}}\rangle$
  - All the parties send their shares to the designated party P_{leader} for that layer.
  - Party P_{leader} proceeds as follows:
    1. It reconstructs all the shares to get individual values $\{z_i^{j,\mathsf{mult}}, z_i^{j,\mathsf{add}}, z_i^{j,\mathsf{relay}}\}_{j\in[\sigma],i\in[w_{m,j}]}$. It then computes the values $z_i^{j,1}, \ldots, z_i^{j,w_m}$ on the outgoing wires from the gates in layer m as follows: For each $j \in [\sigma], i \in [w_{m,j}]$:
       * If gate $g_m^{j,i}$ is a multiplication gate, it sets $z^{j,i} = z_i^{j,\mathsf{mult}}$.
       * If gate $g_m^{j,i}$ is an addition gate, it sets $z^{j,i} = z_i^{j,\mathsf{add}}$.
       * If gate $g_m^{j,i}$ is a relay gate, it sets $z^{j,i} = z_i^{j,\mathsf{relay}}$.
    2. It then computes $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j = \mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$.
    3. For each $j \in [\sigma]$ and $q \in [w_{m+1,j}/\ell]$ each $i \in [\ell]$, let $e_{\mathsf{left}} = \mathsf{LeftInputs}[\ell \cdot (j-1) + i]$ and $e_{\mathsf{right}} = \mathsf{RightInputs}[\ell \cdot (j-1) + i]$, it sets $\overline{\mathbf{z}}^{j,q,\mathsf{left}}[i] = z^{j,e_{\mathsf{left}}}$ and $\overline{\mathbf{z}}^{j,q,\mathsf{right}}[i] = z^{j,e_{\mathsf{right}}}$.
    4. For each $j \in [\sigma], q \in [w_{m+1,j}/\ell]$, it then runs $\mathsf{pshare}(\overline{\mathbf{z}}^{j,q,\mathsf{left}}, D)$ and $\mathsf{pshare}(\overline{\mathbf{z}}^{j,q,\mathsf{right}}, D)$ to obtain shares $[\overline{\mathbf{z}}^{j,q,\mathsf{left}}]$ and $[\overline{\mathbf{z}}^{j,q,\mathsf{right}}]$ respectively. It also sends the respective shares to all parties.
  - For each $j \in [\sigma], q \in [w_{m+1,j}/\ell]$, all parties locally subtract the randomness from these packed secret sharings as follows— $[\mathbf{z}^{j,q,\mathsf{left}}] = [\overline{\mathbf{z}}^{j,q,\mathsf{left}}] - [\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]$ and $[\mathbf{z}^{j,q,\mathsf{right}}] = [\overline{\mathbf{z}}^{j,q,\mathsf{right}}] - [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]$.

Output: The parties output their shares in $[\mathbf{z}^{j,q,\mathsf{left}}]$ and $[\mathbf{z}^{j,q,\mathsf{right}}]$, for each $m \in [d]$, $j \in [\sigma]$ and $q \in [w_{m+1,j}/\ell]$.

Figure 3.6: A Protocol for Layer-wise Circuit Evaluation

3.6 Our Order-C Semi-Honest Protocol

In this section, we describe our semi-honest protocol. All parties get a finite field $\mathbb{F}$ and a layered arithmetic circuit C (of width w and no. of gates $|C|$) over $\mathbb{F}$ that computes the function f on inputs of length n as auxiliary inputs.[3]

Protocol: For each $i \in [n]$, party P_i holds input $x_i \in \mathbb{F}$ and the protocol proceeds as follows:

1. **Input Sharing Phase:** All the parties $\{P_1, \ldots, P_n\}$ collectively invoke $f_{\mathsf{pack-input}}$ as follows – every party P_i for $i \in [n]$, sends each of its inputs to the functionality $f_{\mathsf{pack-input}}$ and records its vector of packed shares $\left\{[\mathbf{x}^{j,q}], [\mathbf{y}^{j,q}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$ of the inputs as received from $f_{\mathsf{pack-input}}$. They set $[\mathbf{z}_1^{j,q,\mathsf{left}}] = [\mathbf{x}^{j,q}]$ and $[\mathbf{z}_1^{j,q,\mathsf{right}}] = [\mathbf{y}^{j,q}]$ for each $j \in [\sigma]$ and $q \in [w_{1,j}/\ell]$.

2. **Circuit Evaluation:** The parties collectively run sub-protocol π_{eval} on input shares $\left\{[\mathbf{z}_1^{j,q,\mathsf{left}}], [\mathbf{z}_1^{j,q,\mathsf{right}}], [\mathbf{z}_1^{j,q,\mathsf{left}}], [\mathbf{z}_1^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$.

3. **Output Reconstruction:** For each $\left\{[\mathbf{z}_{d+1}^{j,q,\mathsf{left}}], [\mathbf{z}_{d+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{d+1,j}/\ell]}$, the parties run the reconstruction algorithm of packed secret sharing to learn the ouput.

We now prove semi-honest security of this protocol.

Lemma 5. *The above protocol is private against a semi-honest adversary that corrupts upto t parties.*

Proof. Let $\mathcal{A}$ be the real adversary. We slightly abuse notation and let $\mathcal{A}$ also denote the set of corrupt parties. Let $\mathcal{H}$ denote the set of honest parties. We construct the simulator Sim as follows. For each $m \in [d]$, let $\{\mathsf{block}_{m,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m, \mathsf{layer_m})$ and $\{\mathsf{block}_{m+1,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m+1, \mathsf{layer_{m+1}})$. Let Unique be as defined in Figure 3.6. Given the output of the protocol and inputs of the honest parties, for each $j \in [\sigma]$, the simulator proceeds as follows:

- **Input Sharing Phase:** It receives the shares $\{[\mathbf{x}^{j,q}]_{\mathcal{A}}, [\mathbf{y}^{j,q}]_{\mathcal{A}}\}_{j\in[\sigma],q\in[w_{q,j}/\ell]}$ that the adversary sends to $f_{\mathsf{pack-input}}$.

[3] For simplicity we assume that each party has only one input. But our protocol can be trivially extended to accommodate scenarios where each party has multiple inputs.

- **Circuit Evaluation:**

  - *Correlated Randomness Generation:* For each $(\mathsf{block}_{a+1}, \mathsf{block}_a) \in \mathsf{Unique}$:

    * For each $i \in \mathcal{H}$, the simulator sends the following random shares

$$\{[\mathbf{s}_i^{q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{s}_i^{q,\mathsf{right}}]_{\mathcal{A}}\}_{q \in [w_{a+1}/\ell]}, \{\langle \mathbf{s}_i^{q,\mathsf{mult}} \rangle_{\mathcal{A}}, \langle \mathbf{s}_i^{q,\mathsf{add}} \rangle_{\mathcal{A}}, \langle \mathbf{s}_i^{q,\mathsf{relay}} \rangle_{\mathcal{A}}\}_{q \in [w_a/\ell]}$$

    to the adversary and receives the following shares from the adversary, for each $i \in \mathcal{A}$.

$$\{[\mathbf{s}_i^{q,\mathsf{left}}]_{\mathcal{H}}, [\mathbf{s}_i^{q,\mathsf{right}}]_{\mathcal{H}}\}_{q \in [w_{a+1}/\ell]}, \{\langle \mathbf{s}_i^{q,\mathsf{mult}} \rangle_{\mathcal{H}}, \langle \mathbf{s}_i^{q,\mathsf{add}} \rangle_{\mathcal{H}}, \langle \mathbf{s}_i^{q,\mathsf{relay}} \rangle_{\mathcal{H}}\}_{q \in [w_a/\ell]}.$$

    * For each $i \in \mathcal{A}$, it uses the above shares to compute the following shares $\{\{[\mathbf{r}_i^{q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_i^{q,\mathsf{right}}]_{\mathcal{A}}\}_{q \in [w_{a+1}/\ell]}, \{\langle \mathbf{r}_i^{q,\mathsf{mult}} \rangle_{\mathcal{A}}, \langle \mathbf{r}_i^{q,\mathsf{add}} \rangle_{\mathcal{A}}, \langle \mathbf{r}_i^{q,\mathsf{relay}} \rangle_{\mathcal{A}}\}_{q \in [w_a/\ell]}\}_{i \in [n-t]}$, where w_a and w_{a+1} are the lengths of blocks block_a and block_{a+1} respectively. It then assigns these shares to different blocks in the circuit based on the configuration of each block. At the end of this step for each $m \in [d], j \in [\sigma]$, the simulator has the following shares:

$$\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\}_{j,q \in [w_{m+1,j}/\ell]}, \{\langle \mathbf{r}_m^{j,q,\mathsf{mult}} \rangle_{\mathcal{A}}, \langle \mathbf{r}_m^{j,q,\mathsf{add}} \rangle_{\mathcal{A}}, \langle \mathbf{r}_m^{j,q,\mathsf{relay}} \rangle_{\mathcal{A}}\}_{q \in [w_{m,j}/\ell]}.$$

  - *Layer-wise Circuit Evaluation:* For each $m \in [d], j \in [\sigma]$:

    * If $P_{\mathsf{leader}} \in \mathcal{H}$, the simulator simulates sending random shares $\left\{[\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q \in [w_{m+1,j}/\ell]}$ to the adversary. It also uses $\left\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q \in [w_{m+1,j}/\ell]}$ to compute shares $\left\{[\mathbf{z}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{z}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q \in [w_{m+1,j}/\ell]}$.

    * Else if $P_{\mathsf{leader}} \in \mathcal{A}$, the simulator simulates sending random shares $\{\langle \mathbf{z}_m^{j,q,\mathsf{mult}} \rangle_{\mathcal{H}}, \langle \mathbf{z}_m^{j,q,\mathsf{add}} \rangle_{\mathcal{H}}, \langle \mathbf{z}_m^{j,q,\mathsf{relay}} \rangle_{\mathcal{H}}\}_{q \in [w_{m,j}/\ell]}$ on behalf of the honest parties to the adversary. Based on the shares $\left\{[\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{H}}, [\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{H}}\right\}_{q \in [w_{m+1,j}/\ell]}$ sent by the adversary and $\left\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q \in [w_{m+1,j}/\ell]}$, the simulator computes

$$\left\{[\mathbf{z}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{z}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}.$$

- **Output Reconstruction:** For each $j \in [\sigma]$, using the output and previously computed shares $\left\{[\mathbf{z}_{d+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{z}_{d+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{d+1,j}/\ell]}$, the simulator computes consistent shares $\left\{[\mathbf{z}_{d+1}^{j,q,\mathsf{left}}]_{\mathcal{H}}, [\mathbf{z}_{d+1}^{j,q,\mathsf{right}}]_{\mathcal{H}}\right\}_{q\in[w_{d+1,j}/\ell]}$ and sends them to the adversary.

View of the adversary generated by the simulator in the correlated randomness generation step is identically distributed to that in the real protocol. Moreover, as shown in [DN07], from super-invertibility property of the Vandermonde matrix, it follows that the $\mathbf{r}_i$ vectors generated by the parties at the end of the this step are random values that are unknown to any individual party or the adversary. Indistinguishability between the view of an adversary in the real protocol and the transcript generated by the simulator in the circuit evaluation step now follows from the privacy of packed secret sharing and the from the fact that the shares sent to the leader are of values that are masked by random values unknown to any party and hence appear completely random to the adversary.

□

Next we calculate the complexity of this protocol.

Complexity of Our Semi-Honest Protocol. For each layer in the protocol, we generate $5 \times$ (width of the layer$/\ell$) packed shares, where $\ell = n/\epsilon$. We have $t = n\left(\frac{1}{2} - \frac{2}{\epsilon}\right)$. In the semi-honest setting, $n - t = n(\frac{1}{2} + \frac{2}{\epsilon})$ of these can be computed with n^2 communication. Therefore, overall the total communication required to generate all the correlated random packed shares is $5 \times |C|2\epsilon^2/(4+\epsilon) = 10|C|\epsilon^2/(4+\epsilon)$.

Additional communication required to evaluate each layer of the circuit is $5n\times$(width of the layer$/\ell$). Therefore, overall the total communication to generate correlated randomness and to evaluate the circuit is $10|C|\epsilon^2/(4+\epsilon)+5|C|\epsilon = \frac{5|C|\epsilon(3\epsilon+4)}{4+\epsilon}$. An additional overhead to generate packed input shares for all inputs is at most $4n|\mathcal{I}|$, where $|\mathcal{I}|$ is the number of inputs to the protocol. Therefore, the total communication complexity is $\frac{5|C|\epsilon(3\epsilon+4)}{4+\epsilon} + 4n|\mathcal{I}|$. As discussed in Section 2.2, using FFT for secret

sharing will yield computation complexity that is $O(\log n)$ times the communication complexity.

3.7 Our Order-C Maliciously Secure Protocol

In this section, we present our maliciously secure Order-C MPC protocols. In addition to the sub protocols/functionalities from Sections 3.4 and 3.5, this construction also depends on some additional sub-protocols/functionalities. In this section, we first present those additional sub-protocols and then proceed to describe our maliciously secure order C MPC protocol.

3.7.1 Generating Random Packed Shares

In this section we describe a natural extension to π_{rand} that allows for generation of random *packed* shares. In our malicious secure protocol, these random values will allow us to efficiently check for linear errors injected by the adversary. We actually require two slightly different functionalities that we will represent in a single ideal functionality, as they are deeply related. The first functionality will generate packed sharings of vectors in which each element is independent. The second functionality will generate packed sharing of vectors in which each element is the same random value. The ideal functionality $f_{\mathsf{pack-rand}}$ is described in Figure 3.7. Since the adversary can choose its own shares in the protocol, similar to Chida et. all [CGH+18], we let adversary send shares of the corrupted parties to the ideal functionality.

We now describe the protocol $\pi_{\mathsf{pack-rand}}$ that securely realizes this functionality $f_{\mathsf{pack-rand}}$. The protocol proceeds as follows:

Auxiliary Inputs Hyper-invertible matrix $\mathbf{H}_{n,n}$

Inputs: The parties do not have any inputs.

Protocol $\pi_{\mathsf{pack-rand}}$**:** The parties proceed as follows:

- If the parties wish to realize the independent mode of $f_{\mathsf{pack-rand}}$, each party P_i (for $i \in [n]$) chooses a random vector $\mathbf{u}_i \in \mathbb{F}^{\ell}$. It runs $\mathsf{pshare}(\mathbf{u}_i, D)$ to receive shares $[\mathbf{u}_i]$. For each $j \in [n]$, it party P_j, its share in $[\mathbf{u}_j]$.

The functionality $f_{\mathsf{pack-rand}}(\{P_1, \ldots, P_n\})$

The n-party functionality $f_{\mathsf{pack-rand}}$, running with parties $\{P_1, \ldots, P_n\}$ and the ideal adversary Sim proceeds as follows:

- Each honest party sends $\mathsf{mode} \in \{\mathsf{independent}, \mathsf{uniform}\}$ to the ideal functionality. If the honest players do not agree, the ideal functionality outputs $\perp$ and aborts.
- The ideal simulator Sim sends U_i for each corrupt party $i \in \mathcal{A}$.
- If $\mathsf{mode} = \mathsf{independant}$, the functionality $f_{\mathsf{pack-rand}}$ chooses a random vector $R \in \mathbb{F}^\ell$ such that each value in $r \in R$ is sampled independently from the field.
- If $\mathsf{mode} = \mathsf{uniform}$, the functionality $f_{\mathsf{pack-rand}}$ chooses a random value $r \in \mathbb{F}$ and sets $R \in \mathbb{F}^\ell$ to be r repeated ℓ times.
- The functionality $f_{\mathsf{pack-rand}}$ sets $[R]_\mathcal{A} = \{U_i\}_{i \in \mathcal{A}}$. It runs $\mathsf{pshare}(R, \mathcal{A}, [R]_\mathcal{A}, T)$ to receive a share R_i for each party P_i.
- It hands each honest party P_j its share R_j.

Figure 3.7: Packed random share generation functionality

- If the parties wish to realize the uniform mode of $f_{\mathsf{pack-rand}}$, each party P_i (for $i \in [n]$) chooses a random element $u_i \in \mathbb{F}$ and computes the vector $\mathbf{u}_i \in \mathbb{F}^\ell$ such that each element is u_i. It runs $\mathsf{pshare}(U_i, T)$ to receive shares $[\mathbf{u}_i]$. For each $j \in [n]$, it party P_j, its share in $[\mathbf{u}_j]$.
- Given shares $([\mathbf{u}_1], \ldots [\mathbf{u}_n])$, the parties compute

$$([\mathbf{r}_1], \ldots, [\mathbf{r}_n]) = \mathbf{H}_{n,n} \cdot ([\mathbf{u}_1], \ldots, [\mathbf{u}_n])$$

- Each party broadcasts its share in $[\mathbf{r}_{n-t+1}], \ldots, [\mathbf{r}_n]$ to the first $t+1$ parties. Those parties locally runs $\mathsf{open}([\mathbf{r}_{n-t+1}]), \ldots, \mathsf{open}([\mathbf{r}_n])$ to check if all the shares lie on the same degree D polynomial.
- If the parties wish to realize the uniform mode of $f_{\mathsf{pack-rand}}$, they additionally check that all $p(e_i)_{i \in [\ell]}$ are the same If these checks succeed, then the parties send "pass" to all other parties, else they send "fail".
- If each of the first $t+1$ parties output "pass", then parties output their shares in $[\mathbf{r}_1], \ldots, [\mathbf{r}_{n-t}]$.

Output: The parties output $[\mathbf{r}_1], \ldots [\mathbf{r}_{n-t}]$.

Lemma 6. *This protocol securely computes $n-t$ instantiations of $f_{\mathsf{pack-rand}}$ with abort in the presence of malicious adversaries who controls t parties.*

The proof of this lemma follows from [DN07].

3.7.2 Checking Equality to Zero

In this section, we discuss the protocol of Chida et.al [CGH+18] to check whether a given sharing is a sharing of the value 0, without revealing any further information on the shared value. We extend this protocol to consider packed secret shares with the natural definition. We describe this functionality in Figure 4.4.

The functionality $f_{\mathsf{checkZero}}(\{P_1, \ldots, P_n\})$

The n-party functionality $f_{\mathsf{checkZero}}$, running with parties $\{P_1, \ldots, P_n\}$ and the ideal adversary Sim receives $[\mathbf{v}]_{\mathcal{H}}$ from the honest parties and uses them to compute $\mathbf{v}$.

- If $\mathbf{v} = 0^\ell$, then $f_{\mathsf{checkZero}}$ sends 0 to the ideal adversary Sim. If Sim responds with reject (resp., accept), then $f_{\mathsf{checkZero}}$ sends reject (resp., accept) to the honest parties.
- If $\mathbf{v} \neq 0^\ell$, then $f_{\mathsf{checkZero}}$ proceeds as follows:
  - With probability $\frac{1}{|\mathbb{F}|}$ it sends accept to the honest parties and ideal adversary Sim.
  - With probability $1 - \frac{1}{|\mathbb{F}|}$ it sends reject to the honest parties and ideal adversary Sim.

Figure 3.8: Random share generation functionality

We now describe the protocol $\pi_{\mathsf{checkZero}}$ that securely realizes this functionality $f_{\mathsf{checkZero}}$. The protocol proceeds as follows:

Inputs: The parties $\{P_i\}_{i \in [n]}$ hold shares $[v]$.

Protocol $\pi_{\mathsf{checkZero}}$: The parties proceed as follows:

- The parties $\{P_1, \ldots, P_n\}$ invoke f_{rand} to obtain sharings $[\mathbf{r}]$.
- The parties $\{P_1, \ldots, P_n\}$ invoke f_{mult} on $[\mathbf{r}]$ and $[\mathbf{v}]$ to obtain $[\mathbf{t}] = [\mathbf{r} \cdot \mathbf{v}]$.

– Each party P_i (for $i \in [n]$) send $\mathbf{t}_i$ to all other parties.

– Each party locally runs $\mathsf{open}([\mathbf{t}])$ on the revealed shares and checks if $\mathbf{t} = 0^\ell$. If so it outputs accept, else, it outputs reject.

We note that the proof of security for this protocol follows similarly to [CGH+18] and hence we omit it here.

3.7.3 Secure Dual Evaluation upto Linear Attacks

Next, we define a subprotocol, which takes packed shares of the actual inputs of the parties and packed shares of the randomized inputs and performs a dual evaluation of the circuit on these sets of inputs. Looking ahead, in our main protocol, this sub-protocol will be directly used for circuit evaluation and for maintaining the invariant that for each intermediate packed shared vector $\mathbf{z}$, the parties also compute shares of both $r\mathbf{z}$.

In [GIP15], Genkin et al. had shown that most packed secret sharing based semi-honest protocols satisfy the following property – when run in the presence of a malicious adversary, any attack strategy of the adversary is limited to simply injecting linear attacks on the outputs of multiplication gates. More precisely, a linear attack on multiplication gates is defined by an arbitrary linear combination of the vectors input to the set of gates.

Definition 6 (Linear Attack). *When multiplying two vectors $\mathbf{a}, \mathbf{b}$ of length ℓ each, a linear attack $\mathbf{L} = (\mathbf{L}_{\mathsf{left}}, \mathbf{L}_{\mathsf{right}})$ specifies linear functions $f_{\mathsf{left}} : \mathbb{F}^\ell \to \mathbb{F}^\ell$ and $f_{\mathsf{right}} : \mathbb{F}^\ell \to \mathbb{F}^\ell$, such that the output vector $\mathbf{c}$ is equal to $\mathbf{c} = \mathbf{a} \odot \mathbf{b} + f_{\mathsf{left}}(\mathbf{a}) + f_{\mathsf{right}}(\mathbf{b})$, where $\odot$ denotes the point-wise multiplication of two vectors.*

An important point to note about these attacks is that the linear attack $\mathbf{L}_{\mathsf{left}}$ is determined based on how $\mathbf{b}$ was secret shared and linear attack $\mathbf{L}_{\mathsf{right}}$ is determined based on how $\mathbf{a}$ was secret shared.

We observe that similar to most semi-honest protocols based on packed secret sharing, our sub-protocol for dual circuit evaluation also has the property that in the presence of a malicious adversary, any attack strategy of the adversary is limited to simply injecting linear attacks on to the

outputs of each gate. We now give a description of this sub-protocol $\pi_{\mathsf{dual-eval}}$.

Inputs: The parties $\{P_i\}_{i\in[n]}$ hold packed secret sharings

$$\left\{[\mathbf{z}_1^{j,q,\mathsf{left}}],[\mathbf{y}_1^{j,q,\mathsf{right}}],[\mathbf{rz}_1^{j,q,\mathsf{left}}],[\mathbf{rz}_1^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}.$$

Dual Circuit Evaluation: The parties collectively run 2 executions of a truncated version of π_{eval} on inputs $\left\{[\mathbf{z}_1^{j,q,\mathsf{left}}],[\mathbf{z}_1^{j,q,\mathsf{right}}],[\mathbf{z}_1^{j,q,\mathsf{left}}],[\mathbf{z}_1^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$, and $\left\{[\mathbf{z}_1^{j,q,\mathsf{left}}],[\mathbf{z}_1^{j,q,\mathsf{right}}],[\mathbf{rz}_1^{j,q,\mathsf{left}}],[\mathbf{rz}_1^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$ respectively, where in the layer $m=d$, the leader locally computes the masked output vectors, but does not secret share it among the other parties.

Output: The parties output their shares in $[\mathbf{z}^{j,q,\mathsf{left}}]$ and $[\mathbf{z}^{j,q,\mathsf{right}}]$, for each $m\in[d-1]$, $j\in[\sigma]$ and $q\in[w_{m+1,j}/\ell]$. P_{leader} outputs $\{\overline{\mathbf{z}}^{j,q,\mathsf{left}},\overline{\mathbf{z}}^{j,q,\mathsf{right}}\}_{j\in[\sigma],q\in[w_{d+1,j}]}$.

Lemma 7. *The protocol in Section 3.7.3 securely evaluates the circuit C on inputs $\mathbf{x},\mathbf{rx}$ up to linear attacks in the presence of a malicious adversary who controls up to t parties.*

Proof Sketch. [GIP15] show that any packed secret sharing based semi-honest protocol that satisfies the following properties, we know that any packed secret sharing based semi-honest protocol that satisfies the following three properties is secure against an adversary upto linear attacks:

- *T-randomization*: The messages sent by the honest parties to the corrupt parties (except in the last round), only depend on the randomness of the parties and not on their actual inputs.
- *Structure of the Last Round*: During the last round, only one party computes the output vector $\mathbf{z}$, as follows: let $F_{\mathcal{H}}$ and $F_{\mathcal{A}}$ be two linear functions, such that $z=F_{\mathcal{H}}(\mathsf{lmsg}_{\mathcal{H}})+F_{\mathcal{A}}(\mathsf{lmsg}_{\mathcal{A}})$, where $\mathsf{lmsg}_{\mathcal{H}}$ are the messages sent by the honest parties in the last round and $\mathsf{lmsg}_{\mathcal{A}}$ are the messages sent by the corrupt parties in the last round.
- *Privacy of the last round*: The distribution of the messages $\mathsf{lmsg}_{\mathcal{H}}$ sent by the honest parties in the last round are uniform, conditioned on $F_{\mathcal{H}}(\mathsf{lmsg}_{\mathcal{H}})=\mathbf{z}-F_{\mathcal{A}}(\mathsf{lmsg}_{\mathcal{A}})$.

The first property is trivially satisfied by our sub-protocol — indeed, Genkin's thesis [Gen16] already shows that semi-honest [DIK10] satisfies the first property, and those arguments generalize to our sub-protocol in a straightforward way. The second property is also easy to verify, indeed the masked output vector is computed by P_{leader} in our subprotocol, by running the reconstruction algorithm of packed secret sharing, which is a linear function. The third property is also satisfied by our protocol, since the shares sent by the parties in the last round to P_{leader} correspond to shares for a degree $n-1$ polynomial, the shares of the honest parties are uniformly distributed given the output and the shares of the corrupt parties. As a result, our protocol securely evaluates C on inputs $\mathbf{x}, \mathbf{rx}$ up to linear attacks. □

3.7.4 Secure Multiplication upto Linear Attacks

In this section we describe a semi-honest secure multiplication protocol for packed secret shares that is secure up to linear attacks. We require this functionality to realize $f_{\mathsf{checkZero}}$ for packed secret sharing and setting up the randomized protocol execution for malicious security. The ideal functionality for $f_{\mathsf{pack-mult}}$ is given in Figure 3.9.

The functionality $f_{\mathsf{pack-mult}}(\{P_1, \dots, P_n\})$

The functionality $f_{\mathsf{pack-mult}}$, running with a set of parties $\{P_1, \dots, P_n\}$ and the ideal adversary Sim proceeds as follows:

- Upon receiving $[\mathbf{x}]_{\mathcal{H}}$ and $[\mathbf{y}]_{\mathcal{H}}$ from the honest parties, the ideal functionality $f_{\mathsf{pack-mult}}$ reconstructs computes $\mathbf{x} = \{x_i\}_{i \in [\ell]}, \mathbf{y} = \{y_i\}_{i \in [\ell]}$. The simulator also computes shares $[\mathbf{x}]_{\mathcal{A}}$ and $[\mathbf{y}]_{\mathcal{A}}$ and sends them to the adversary.
- Upon receiving Linear error $\mathbf{L} : \mathbb{F}^{2\ell} \to \mathbb{F}^{\ell}$ and $\{\mathbf{u}_i\}_{i \in \mathcal{A}}$ from the ideal adversary Sim, functionality $f_{\mathsf{pack-mult}}$ defines $\mathbf{z} = \mathbf{x} \cdot \mathbf{y} + \mathbf{L}(\mathbf{x}, \mathbf{y})$ and $[\mathbf{z}]_{\mathcal{A}} = \{\mathbf{u}_i\}_{i \in \mathcal{A}}$. It then runs $\mathsf{pshare}(\mathbf{z}, \mathcal{A}, [\mathbf{z}]_{\mathcal{A}}, D)$ to obtain a share $\mathbf{z}_j$ for each party P_j.
- The ideal functionality $f_{\mathsf{pack-mult}}$ hands each honest party P_j its share $\mathbf{z}_j$.

Figure 3.9: Secure Multiplication Up to Linear Attack functionality

We now describe the protocol $\pi_{\mathsf{pack-mult}}$ that securely realizes this functionality $f_{\mathsf{pack-mult}}$ (Figure 3.9). The protocol proceeds as follows:

Inputs: The parties $\{P_i\}_{i\in[n]}$ hold shares $[\mathbf{x}], [\mathbf{y}]$.

Protocol $\pi_{\mathsf{pack-mult}}$**:** The parties proceed as follows:

- The parties $\{P_1, \ldots, P_n\}$ locally compute $\langle \mathbf{x} \cdot \mathbf{y} \rangle = [\mathbf{x}] \cdot [\mathbf{y}]$
- the parties invoke $f_{\mathsf{pack-rand}}$ in independent mode to obtain packed secret shares $[\mathbf{r}]$ and $\langle \mathbf{r} \rangle$ for a random, independent vector $\mathbf{r}$.
- The parties the locally compute $\langle \mathbf{x} \cdot \mathbf{y} \rangle - \langle \mathbf{r} \rangle$ and send the resulting shares to the designated party P_{leader}.
- Party P_{leader} reconstructs all the values $\mathbf{x} \cdot \mathbf{y} - \mathbf{r}$. Party P_{leader} then generates a degree D sharing of this vector $[\mathbf{x} \cdot \mathbf{y} - \mathbf{r}]$ and send the resulting shares to all players
- Players locally compute $[\mathbf{z}] = [\mathbf{x} \cdot \mathbf{y} - \mathbf{r}] + [\mathbf{r}]$

Output: The parties output $[x \cdot y]$

Lemma 8. *This protocol securely computes $f_{\mathsf{pack-mult}}$ up to linear attacks in the $f_{\mathsf{pack-rand}}$-hybrid model, in the presence of malicious adversaries who controls t parties.*

Since this protocol is identical to the multiplication protocol of [DIK10], the proof of this lemma follows from the security proof given in [GIP15].

3.7.5 Maliciously Secure Protocol

We now describe a protocol that achieves security with abort against malicious corruptions.

Auxiliary Inputs: A finite field $\mathbb{F}$ and a layered arithmetic circuit C (of width w and $|C|$ gates) over $\mathbb{F}$ that computes the function f on inputs of length n.

Inputs: For each $i \in [n]$, party P_i holds input $x_i \in \mathbb{F}$.

Protocol: (Throughout the protocol, if any party receives $\perp$ as output from a call to a sub-functionality, then it sends $\perp$ to all other parties, outputs $\perp$ and halts):

1. **Secret-Sharing Inputs:** All the parties $\{P_1, \ldots, P_n\}$ collectively invoke $f_{\mathsf{pack-input}}$ as follows — every party P_i for $i \in [n]$, sends each of its input x_i to functionality $f_{\mathsf{pack-input}}$. and records its vector of packed shares $\left\{[\mathbf{x}^{j,q}], [\mathbf{y}^{j,q}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$ of the inputs as received from $f_{\mathsf{pack-input}}$. They set $[\mathbf{z}_1^{j,q,\mathsf{left}}] = [\mathbf{x}^{j,q}]$ and $[\mathbf{z}_1^{j,q,\mathsf{right}}] = [\mathbf{y}^{j,q}]$ for each $j \in [\sigma]$ and $q \in [w_{1,j}/\ell]$.

2. **Pre-processing:**

   – Random Input Generation: The parties invoke $f_{\mathsf{pack-rand}}$ on mode uniform to receive packed sharings $[\mathbf{r}]$ of a vector $\mathbf{r}$, of the form $\mathbf{r} = (r, \ldots, r)$.

   – The parties also invoke $f_{\mathsf{pack-rand}}$ on mode independent to receive packed sharings $\{[\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}], [\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}]\}_{m\in[d],j\in[\sigma],q\in[w_{m,j}/\ell]}$ of random vectors $\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}, \boldsymbol{\alpha}_m^{j,q,\mathsf{right}}$.

   – Randomizing Inputs: For each packed input sharing $[\mathbf{z}_1^{j,q,\mathsf{left}}], [\mathbf{z}_1^{j,q,\mathsf{right}}]$ (for $j \in [\sigma], q \in [w_{1,j}/\ell]$), the parties invoke f_{mult}, on $[\mathbf{z}_1^{j,q,\mathsf{right}}]$ and $[\mathbf{r}]$ to receive $[\mathbf{rz}_1^{j,q,\mathsf{left}}]$ and on $[\mathbf{z}_1^{j,q,\mathsf{right}}]$ and $[\mathbf{r}]$ to receive $[\mathbf{rz}_1^{j,q,\mathsf{right}}]$.

3. **Dual Circuit Evaluation:** The parties run $\pi_{\mathsf{dual-eval}}$ on inputs $\left\{[\mathbf{z}_1^{j,q,\mathsf{left}}], [\mathbf{y}_1^{j,q,\mathsf{right}}], [\mathbf{rz}_1^{j,q,\mathsf{left}}], [\mathbf{rz}_1^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$ to obtain shares $\left\{[\mathbf{z}_{m+1}^{j,q,\mathsf{left}}], [\mathbf{z}_{m+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{m,j}/\ell]}$ and $\left\{[\mathbf{rz}_{m+1}^{j,q,\mathsf{left}}], [\mathbf{rz}_{m+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{m,j}/\ell]}$ for each $m \in [d-1]$ and the leader party additionally receives the masked output vectors for the last layer. The parties then compute the last two-steps of π_{eval} for the last layer. i.e., the leader party then pack secret shares these vectors among the other parties and all the parties subtract their shares of the random masks from these packed secret shares to obtain shares $\left\{[\mathbf{z}_{d+1}^{j,q,\mathsf{left}}], [\mathbf{z}_{d+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{d,j}/\ell]}$ and $\left\{[\mathbf{rz}_{d+1}^{j,q,\mathsf{left}}], [\mathbf{rz}_{d+1}^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma],q\in[w_{d,j}/\ell]}$.

4. **Verification Step:** Each party does the following:

   (a) For each $m \in [d+1]$, $j \in [\sigma]$,$q \in [w_{m,j}/\ell]$, the parties invoke f_{mult} on their packed shares $([\mathbf{z}_m^{j,q,\mathsf{left}}], [\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}])$, $([\mathbf{rz}_m^{j,q,\mathsf{left}}], [\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}])$, $([\mathbf{z}_m^{j,q,\mathsf{right}}], [\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}])$ and $([\mathbf{rz}_m^{j,q,\mathsf{right}}],$

$[\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}]$), and locally compute. [4]

$$[\mathbf{v}] = \sum_{m\in[d+1]} \sum_{j\in[\sigma],q\in[w_{m,j}/\ell]} [\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}][\mathbf{rz}_m^{j,q,\mathsf{left}}] + [\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}][\mathbf{rz}_m^{j,q,\mathsf{right}}]$$

$$[\mathbf{u}] = \sum_{m\in[d+1]} \sum_{j\in[\sigma],q\in[w_{m,j}/\ell]} [\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}][\mathbf{z}_m^{j,q,\mathsf{left}}] + [\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}][\mathbf{z}_m^{j,q,\mathsf{right}}]$$

(b) The parties open shares $[\mathbf{r}]$ to reconstruct $\mathbf{r} = (r, \ldots, r)$.

(c) Each party then locally computes $[\mathbf{t}] = [\mathbf{v}] - r[\mathbf{u}]$

(d) The parties invoke $f_{\mathsf{checkZero}}$ on $[\mathbf{t}]$. If $f_{\mathsf{checkZero}}$ outputs reject, the output of the parties is $\perp$. Else, if it outputs accept, then the parties proceed.

5. **Output Reconstruction:** For each output vector, the parties run the reconstruction algorithm of packed secret sharing to learn the output. If the reconstruction algorithm outputs $\perp$, then the honest parties output $\perp$ and halt.

3.8 Security Proof for our Maliciously Secure Protocol

Lemma 9. *If $\mathcal{A}$ sends a non-zero linear attack value in any of the calls to f_{mult} or $f_{\mathsf{dual-eval}}$ in the execution of the protocol given in Section 3.7.5, then the vector $\mathbf{t}$ in the verification stage equals a 0-vector with probability less that $2/|\mathbb{F}|$.*

Proof. A malicious adversary can carry out linear attacks on f_{mult} and $\pi_{\mathsf{dual-eval}}$, meaning that the adversary can add an arbitrary linear combination of the input wires of a gate to the value on its outgoing wire. We show that the technique used by Chida et al., for detecting addditive errors can be used in the packed secret sharing setting to detect linear attacks. Since we perform a check on packed shares as opposed to regular shares, our check can be viewed as ℓ parallel checks at the end. We essentially end up computing ℓ different linear combinations of approximately $2|C|/\ell$ values. Given our description of $f_{\mathsf{checkZero}}$, it is clear that if any of these checks

[4] We remark that for notational convinience we describe this step as consisting of $4|C|/\ell$ multiplications (and hence these many degree reduction steps), it can be done with just two degree reduction step, where the parties first locally multiply and add their respective shares to compute $\langle\mathbf{v}\rangle$ and $\langle\mathbf{u}\rangle$ and then communicate to obtain shares of $[\mathbf{v}]$ and $[\mathbf{u}]$ respectively.

fail, $f_{\mathsf{checkZero}}$ will output $\perp$. Therefore, for exact probability calculation, we bound the probability of the adversary injecting errors and getting away in any one of the linear combinations. More specifically, we consider the linear combination over the first elements in each packed sharing output. For each $m \in [d+1]$ in the circuit, our protocol generates $4w_m/\ell$ packed shares of the form $\left\{[\mathbf{z}_m^{j,q,\mathsf{left}}], [\mathbf{z}_m^{j,q,\mathsf{right}}], [\mathbf{z}_m^{j,q,\mathsf{left}}], [\mathbf{rz}_m^{j,q,\mathsf{right}}]\right\}_{j\in[\sigma], q\in[w_{d,j}/\ell]}$. We simplify the notation and let the set of packed shares on each layer m, be of the form $\{[\mathbf{z}_m^q], [\mathbf{rz}_m^q]\}_{q\in[w_m/\ell]}$. Similarly, we use $\boldsymbol{\alpha}_m^q$ to denote the $\boldsymbol{\alpha}$ vectors corresponding to these packed secret sharings.

Finally, we slightly abuse notation and let z_m^q denote the first element in the vector $\mathbf{z}_m^q$ and α_m^q to denote the first element in the vector $\boldsymbol{\alpha}_m^q$.

We use different variables to denote the additive errors that the adversary can inject on each of these computations.

- Let $\mathbf{F}_m^q : \mathbb{F}^\ell \to \mathbb{F}$ be the linear error function induced as a result of operating on incorrectly computed shares of $\mathbf{z}_m^q$. For instance, when the vectors $\mathbf{z}_m^q$ and $\boldsymbol{\alpha}_m^q$ are multiplied, a linear error of the form $\mathbf{F}_m^q(\boldsymbol{\alpha}_m^q)$ is induced on the first element of the output vector. We note that since $\boldsymbol{\alpha}_m^q$ in our protocol is guaranteed to be honestly secret shared, no error of the form $\mathbf{L}(\mathbf{z}_m^q)$ is induced when multiplying $\boldsymbol{\alpha}_m^q$ and $\mathbf{z}_m^q$.
- Similarly, let $\mathbf{G}_m^q : \mathbb{F}^\ell \to \mathbb{F}$ be the linear error function induced as a result of operating on incorrectly computed shares of $\mathbf{rz}_m^q$.
- We let f_m^q be the resultant linear error on z_m^q and g_m^q be the resultant linear error on rz_m^q. We note that these errors are not arbitrary values but a linear combination of the input values to the gates in layer m.

Recall that if every party behaves honestly, then

$$u = \sum_{m\in[d+1]} \sum_{q\in[w_m/\ell]} \alpha_m^q z_m^q \text{ and } v = \sum_{m\in[d+1]} \sum_{m\in[w_m/\ell]} \alpha_q(rz_m^q)$$

We would like to check if $ru = v$, ie.

$$r \sum_{m \in [d+1]} \sum_{q \in [w_m/\ell]} \alpha_m^q z_m^q = \sum_{m \in [d+1]} \sum_{q \in [w_m/\ell]} \alpha_q (r z_m^q)$$

This is trivially true, if no errors were introduced by the adversary at any step. Accounting for all the linear errors that the adversary might introduce, we get

$$ru = r \left(\sum_{m \in [d+1]} \sum_{q \in [w_m/\ell]} \alpha_m^q (z_m^q + f_m^q) + \mathbf{F}_m^q(\boldsymbol{\alpha}_m^q) \right)$$

$$v = \left(\sum_{m \in [d+1]} \sum_{q \in [w_m/\ell]} \alpha_m^q (r z_m^q + g_m^q) + \mathbf{G}_m^q(\boldsymbol{\alpha}_m^q) \right)$$

We want to calculate the probability that the following equation holds, i.e.,

$$r \left[\left(\sum_{m \in [d+1]} \sum_{q \in [w_m/\ell]} \alpha_m^q (z_m^q + f_m^q) + \mathbf{F}_m^q(\boldsymbol{\alpha}_m^q) \right) \right] = \left(\sum_{m \in [d+1]} \sum_{q \in [w_m/\ell]} \alpha_m^q (r z_m^q + g_m^q) + \mathbf{G}_m^q(\boldsymbol{\alpha}_m^q) \right)$$

In other words,

$$\sum_{m \in [d+1]} \sum_{q \in [w_m/\ell]} \alpha_m^q (r f_m^q - g_m^q) = \sum_{m \in [d+1]} \sum_{q \in [w_m/\ell]} \mathbf{G}_m^q(\boldsymbol{\alpha}_m^q) - r\mathbf{F}_m^q(\boldsymbol{\alpha}_m^q)$$

We now consider the following cases:

– **Case 1:** All the inputs, intermediate wire computations and $\alpha's$ were honestly secret shared and computed. And the errors were only introduced during the verification step. Since the verification step in this case corresponds to multiplying honestly secret shared vectors, the only kind of errors that the adversary can introduce in the case are arbitrary additive errors, that are not correlated to any of the input values. Let d_u be the cumulative additive error on the computation of u, ans d_v be the cumulative additive error on the computaiton of v. We can re-write the above equation as $0 = \sum_{m \in [d+1]} \sum_{q \in [w_m/\ell]} \mathbf{G}_m^q(\boldsymbol{\alpha}_m^q) - r\mathbf{F}_m^q(\boldsymbol{\alpha}_m^q) = d_u - r d_v$.

  Since r is sampled uniformly, the probability that $d_u - r d_v = 0$ is $1/|\mathbb{F}|$, if either $d_u \neq 0$ or $d_v \neq 0$.

– **Case 2:** $\exists m \in [d], \exists q \in [w_m/\ell]$, such that the output $\mathbf{z}_m^q$ was not correctly secret shared. Let m_0 be the smallest such m and q_0 be the smallest such q. We want to calculate the probability that the following equation holds, i.e.,

$$\begin{aligned}
&\mathbf{G}_{m_0}^{q_0}(\boldsymbol{\alpha}_{m_0}^{q_0}) - r\mathbf{F}_{m_0}^{q_0}(\boldsymbol{\alpha}_m^q) \\
&= \alpha_{m_0}^{q_0}(rf_{m_0}^{q_0} - g_{m_0}^{q_0}) + \sum_{m \in [d+1], m \neq m_0} \sum_{q \in [w_m/\ell]} r\alpha_m^q(f_m^q - g_m^q) - \mathbf{G}_m^q(\boldsymbol{\alpha}_m^q) + r\mathbf{F}_m^q(\boldsymbol{\alpha}_m^q)
\end{aligned}$$

  – If $\mathbf{G}_{m_0}^{q_0}(\boldsymbol{\alpha}_{m_0}^{q_0}) - r\mathbf{F}_{m_0}^{q_0}(\boldsymbol{\alpha}_m^q) \neq 0$: Since all the $\alpha' s$ (including $\alpha_{m_0}^{q_0}$) are generated honestly and are unknown to the adversary, the above equality holds only with probability $1/|\mathbb{F}|$.

  – If $\mathbf{G}_{m_0}^{q_0}(\boldsymbol{\alpha}_{m_0}^{q_0}) - r\mathbf{F}_{m_0}^{q_0}(\boldsymbol{\alpha}_m^q) = 0$: Since r is sampled uniformly at random, this only happens with probability $1/|\mathbb{F}|$.

Hence, overall the probability that that the view generated by the simulator in Case 2 is distinguishable from the view in the real execution is at most

$$\frac{1}{|\mathbb{F}|} + \left(1 - \frac{1}{|\mathbb{F}|}\right)\frac{1}{|\mathbb{F}|} < \frac{2}{|\mathbb{F}|}$$

In both cases, the probability of distinguishability is upper bounded by $\frac{2}{|\mathbb{F}|}$. □

Operating over Smaller fields. This protocol works for fields that are large enough such that $\frac{2}{|\mathbb{F}|}$ is an acceptable probability of an adversary cheating. In cases where it might be desirable to instead work in a smaller field, we can use the same approach as used by Chida et al. [CGH+18]. In particular, instead of having a single randomized evaluation of the circuit w.r.t. r, we can generate shares for δ random values $r_1, \ldots, r_\delta$ (such that $(\frac{2}{|\mathbb{F}|})^\delta$ is negligible) and run multiple randomized evaluations of the circuit and verification steps for each r_i. Since each r is independently sampled and their corresponding verification procedures are also independent, this will yield a cheating probability of at most $(\frac{2}{|\mathbb{F}|})^\delta$, as required.

Given this lemma, we now prove the following Theorem.

Theorem 1. *Let k be a statistical security parameter, and let $\mathbb{F}$ be a finite field such that $(3/|\mathbb{F}|)^{\delta} \leq 2^{-k}$, for some $\delta \geq 1$. Let f be an n-party functionality over $\mathbb{F}$. Then, there exists a protocol in the $(f_{\mathsf{pack-input}}, f_{\mathsf{pack-rand}}, f_{\mathsf{dual-eval}}, f_{\mathsf{mult}}, f_{\mathsf{checkZero}})$-hybrid model with statistical error 2^{-k}, in the presence of a malicious adversary controlling t parties.*

Proof. For each $m \in [d]$, let $\{\mathsf{block}_{m,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m, \mathsf{layer_m})$ and $\{\mathsf{block}_{m+1,j}\}_{j\in[\sigma]} \leftarrow \mathsf{part}(m+1, \mathsf{layer_{m+1}})$. Let $\mathcal{A}$ be an adversary in the real world. As before we use $\mathcal{A}$ to also denote the set of corrupted parties. The simulator Sim in the ideal world initializes a variable $\mathsf{flag} = 0$ and proceeds as follows:

1. **Input Sharing:** Sim receives from $\mathcal{A}$ the set of corrupted inputs and the shares of corrupted parties $\left\{[\mathbf{z}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{z}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{j\in[\sigma],q\in[w_{1,j}/\ell]}$ that the adversary sends to the $f_{\mathsf{pack-input}}$ functionality. It reconstructs these inputs and saves them.

2. **Preprocessing:** Simulator receives shares $[\mathbf{r}]_{\mathcal{A}}$ and $\{[\boldsymbol{\alpha}_m^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\boldsymbol{\alpha}_m^{j,q,\mathsf{right}}]_{\mathcal{A}}\}_{m\in[d],j\in[\sigma],q\in[w_{m,j}/\ell]}$ of the corrupted parties that the adversary sends to $f_{\mathsf{pack-rand}}$.

3. **Randomization of inputs:** For each $j \in [\sigma], q \in [2w_{1,j}/\ell]$, the simulator Sim plays the role of f_{mult} in the multiplication of vectors $\mathbf{z}^{j,q,\mathsf{left}}$ and $\mathbf{z}^{j,q,\mathsf{right}}$ with $[\mathbf{r}]$. Specifically, Sim hands the corrupted parties shares in $[\mathbf{r}]$, $\mathbf{z}^{j,q,\mathsf{left}}$ and $\mathbf{z}^{j,q,\mathsf{left}}$ to the adversary. Upon receiving the linear error function and the corrupted parties shares of the resulting vector, the simulator stores all the corrupted parties' shares. If any linear error function was received, it sets $\mathsf{flag} = 1$.

4. **Dual Circuit Evaluation:** As discussed in the main protocol, dual circuit evaluation requires the parties to run π_{eval} twice – on actual inputs and on randomized inputs. Here we describe how the transcript for evaluation on actual inputs it simulated. The transcript on randomized inputs is simulated in almost exactly the same way, and hence we omit it here.

   – *Correlated Randomness Generation for circuit evaluation on actual inputs:*

     For each $(\mathsf{block}_{a+1}, \mathsf{block}_a) \in \mathsf{Unique}$:

– For each $i \in \mathcal{H}$, the simulator sends the following random shares

$$\{[\mathbf{s}_i^{q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{s}_i^{q,\mathsf{right}}]_{\mathcal{A}}\}_{q \in [w_{a+1}/\ell]}, \{\langle \mathbf{s}_i^{q,\mathsf{mult}} \rangle_{\mathcal{A}}, \langle \mathbf{s}_i^{q,\mathsf{add}} \rangle_{\mathcal{A}}, \langle \mathbf{s}_i^{q,\mathsf{relay}} \rangle_{\mathcal{A}}\}_{q \in [w_a/\ell]}$$

to the adversary and receives the following shares from the adversary, for each $i \in \mathcal{A}$.

$$\{[\mathbf{s}_i^{q,\mathsf{left}}]_{\mathcal{H}}, [\mathbf{s}_i^{q,\mathsf{right}}]_{\mathcal{H}}\}_{q \in [w_{a+1}/\ell]}, \{\langle \mathbf{s}_i^{q,\mathsf{mult}} \rangle_{\mathcal{H}}, \langle \mathbf{s}_i^{q,\mathsf{add}} \rangle_{\mathcal{H}}, \langle \mathbf{s}_i^{q,\mathsf{relay}} \rangle_{\mathcal{H}}\}_{q \in [w_a/\ell]}.$$

– The simulator computes $\mathsf{LeftInputs}, \mathsf{RightInputs} = \mathsf{WireConfiguration}(\mathsf{block}_{a+1}, \mathsf{block}_a)$. For each $q \in [w_{a+1}/\ell]$ and for each $k \in [\ell]$, it sets $e_{\mathsf{left}} = \mathsf{LeftInputs}[(q-1)\ell + i]$ and $e_{\mathsf{right}} = \mathsf{RightInputs}[(q-1)\ell + i]$. For each $i \in \mathcal{A}$, the simulator checks if

$$\mathbf{s}_i^{q,\mathsf{left}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{left}}/\ell \rfloor, \mathsf{GateType}_k}[e_{\mathsf{left}} - \lfloor e_{\mathsf{left}}/\ell \rfloor]$$

$$\mathbf{s}_i^{q,\mathsf{right}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{right}}/\ell \rfloor, \mathsf{GateType}_k}[e_{\mathsf{right}} - \lfloor e_{\mathsf{right}}/\ell \rfloor]$$

where $\mathsf{GateType}_k = \mathsf{mult}$ if gate k in block a is a multiplication gate, else if it is an addition gate then $\mathsf{GateType}_k = \mathsf{add}$ and for relay gates, $\mathsf{GateType}_k = \mathsf{relay}$. If any of these checks fail for any $i \in \mathcal{A}$, the simulator sets $\mathsf{flag} = 1$.

– For each $i \in \mathcal{A}$, it uses the above shares to compute the following shares

$$\{\{[\mathbf{r}_i^{q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_i^{q,\mathsf{right}}]_{\mathcal{A}}\}_{q \in [w_{a+1}/\ell]}, \{\langle \mathbf{r}_i^{q,\mathsf{mult}} \rangle_{\mathcal{A}}, \langle \mathbf{r}_i^{q,\mathsf{add}} \rangle_{\mathcal{A}}, \langle \mathbf{r}_i^{q,\mathsf{relay}} \rangle_{\mathcal{A}}\}_{q \in [w_a/\ell]}\}_{i \in [n-t]},$$

where w_a and w_{a+1} are the lengths of blocks block_a and block_{a+1} respectively.

– It then assigns these shares to different blocks in the circuit based on the configuration of each block. At the end of this step for each $m \in [d], j \in [\sigma]$, the simulator has the following shares:

$$\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\}_{j,q \in [w_{m+1,j}/\ell]}, \{\langle \mathbf{r}_m^{j,q,\mathsf{mult}} \rangle_{\mathcal{A}}, \langle \mathbf{r}_m^{j,q,\mathsf{add}} \rangle_{\mathcal{A}}, \langle \mathbf{r}_m^{j,q,\mathsf{relay}} \rangle_{\mathcal{A}}\}_{q \in [w_{m,j}/\ell]}.$$

– *Circuit evaluation on actual inputs:* The simulator computes $\mathsf{LeftInputs}_j, \mathsf{RightInputs}_j =$

$\mathsf{WireConfiguration}(\mathsf{block}_{m+1,j}, \mathsf{block}_{m,j})$. For each $q \in [w_{m+1,j}/\ell]$ and for each $k \in [\ell]$, it sets $e_{\mathsf{left}} = \mathsf{LeftInputs}[(q-1)\ell + i]$ and $e_{\mathsf{right}} = \mathsf{RightInputs}[(q-1)\ell + i]$. For each $i \in \mathcal{A}$, the simulator check if

$$\mathbf{s}_i^{q,\mathsf{left}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{left}}/\ell \rfloor, \mathsf{GateType}_k}[e_{\mathsf{left}} - \lfloor e_{\mathsf{left}}/\ell \rfloor]$$

$$\mathbf{s}_i^{q,\mathsf{right}}[k] = \mathbf{s}_i^{\lfloor e_{\mathsf{right}}/\ell \rfloor, \mathsf{GateType}_k}[e_{\mathsf{right}} - \lfloor e_{\mathsf{right}}/\ell \rfloor]$$

where $\mathsf{GateType}_k = \mathsf{mult}$ if gate k on layer m is a multiplication gate, else if it is an addition gate then $\mathsf{GateType}_k = \mathsf{add}$ and for relay gates, $\mathsf{GateType}_k = \mathsf{relay}$. If any of these checks fail for any $i \in \mathcal{A}$, the simulator sets $\mathsf{flag} = 1$.

– For each $i \in \mathcal{A}$, it uses the above shares to compute the following shares

$$\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\}_{j,q\in[w_{m+1,j}/\ell]}, \{\langle \mathbf{r}_m^{j,q,\mathsf{mult}} \rangle_{\mathcal{A}}, \langle \mathbf{r}_m^{j,q,\mathsf{add}} \rangle_{\mathcal{A}}, \langle \mathbf{r}_m^{j,q,\mathsf{relay}} \rangle_{\mathcal{A}}\}_{q\in[w_{m,j}/\ell]}.$$

– If $P_{\mathsf{leader}} \in \mathcal{H}$:

(a) Simulator receives shares $\{\langle \mathbf{z}_m^{j,q,\mathsf{mult}} \rangle_{\mathcal{A}}, \langle \mathbf{z}_m^{j,q,\mathsf{add}} \rangle_{\mathcal{A}}, \langle \mathbf{z}_m^{j,q,\mathsf{relay}} \rangle_{\mathcal{A}}\}_{q\in[w_{m,j}/\ell]}$ from the adversary.

(b) For each $i \in \mathcal{A}$, the simulator checks if $\langle \mathbf{z}_m^{j,q,\mathsf{mult}} \rangle_i = [\mathbf{z}_m^{j,q,\mathsf{left}}]_i \cdot [\mathbf{z}_m^{j,q,\mathsf{right}}]_i + \langle \mathbf{r}_m^{j,q,\mathsf{mult}} \rangle_i$ and $\langle \mathbf{z}_m^{j,q,\mathsf{add}} \rangle_i = [\mathbf{z}_m^{j,q,\mathsf{left}}]_i + [\mathbf{y}_m^{j,q,\mathsf{right}}]_i + \langle \mathbf{r}^{j,q,\mathsf{add}} \rangle_i$ and $\langle \mathbf{z}_m^{j,q,\mathsf{relay}} \rangle_i = [\mathbf{z}_m^{j,q,\mathsf{left}}]_i + \langle \mathbf{r}_m^{j,q,\mathsf{relay}} \rangle_i$. If any of these checks fail, the simulator sets $\mathsf{flag} = 1$.

(c) Simulator simulates sending random shares $\left\{[\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$ to the adversary.

(d) Simulator uses $\left\{[\mathbf{r}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{r}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$ to compute shares $\left\{[\mathbf{z}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{A}}, [\mathbf{z}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$.

– Else if $P_{\mathsf{leader}} \in \mathcal{A}$:

(a) The simulator simulates sending random shares $\{\langle \mathbf{z}_m^{j,q,\mathsf{mult}} \rangle_{\mathcal{H}}, \langle \mathbf{z}_m^{j,q,\mathsf{add}} \rangle_{\mathcal{H}}$ and $\langle \mathbf{z}_m^{j,q,\mathsf{relay}} \rangle_{\mathcal{H}}\}_{q\in[w_{m,j}/\ell]}$ on behalf of the honest parties to the adversary.

(b) The simulator uses the shares $\left\{[\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{left}}]_{\mathcal{H}}, [\overline{\mathbf{z}}_{m+1}^{j,q,\mathsf{right}}]_{\mathcal{H}}\right\}_{q\in[w_{m+1,j}/\ell]}$ sent by the adver-

sary to reconstruct $\overline{\mathbf{z}}^{j,q,\mathsf{left}}_{m+1}$ and $\overline{\mathbf{z}}^{j,q,\mathsf{right}}_{m+1}$ and checks if these are consistent with the previously sent and computed shares. If not, it sets $\mathsf{flag} = 1$.

(c) Finally, the simulator uses $\overline{\mathbf{z}}^{j,q,\mathsf{left}}_{m+1}$, $\overline{\mathbf{z}}^{j,q,\mathsf{right}}_{m+1}$ and $\left\{[\mathbf{r}^{j,q,\mathsf{left}}_{m+1}]_{\mathcal{A}}, [\mathbf{r}^{j,q,\mathsf{right}}_{m+1}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$, to compute $\left\{[\mathbf{z}^{j,q,\mathsf{left}}_{m+1}]_{\mathcal{A}}, [\mathbf{z}^{j,q,\mathsf{right}}_{m+1}]_{\mathcal{A}}\right\}_{q\in[w_{m+1,j}/\ell]}$.

5. **Verification Step:** The simulator simulates the honest parties sending their shares in the opening of $[\mathbf{r}]$ to the adversary and receives the shares that the adversary sends to the honest parties in this open. If any honest party would abort, then the simulator simulates it by sending $\perp$ to all parties, and to the trusted functionality and halts. Finally, Sim simulates $f_{\mathsf{checkZero}}$ as follows, If $\mathsf{flag} = 1$, then Sim simulates $f_{\mathsf{checkZero}}$ by sending reject and then all honest parties sending $\perp$. Otherwise, Sim proceeds to the next step.

6. **Output Reconstruction:** If no abort had occurred, Sim sends the inputs of the adversary that it had extracted from the input sharing phase to the ideal functionality computing f. It receives back the output from the ideal functionality. The simulator then computes the shares of the honest parties using this output and the shares of the corrupt parties (that it can compute based on the information it has). It sends these shares to the adversary as part of the output reconstruction phase.

   It then receives messages from the adversary. It uses these messages to reconstruct the output for the honest party. If for any honest party, this reconstruction fails, it sends $\perp$ along with the identity of the honest party to the ideal functionality, signaling it to send $\perp$ to that party.

Overall, the view of the adversary in the ideal world is identical to its view in the real world, except with $3/|\mathbb{F}|$ probability. Up until the correlated randomness generation step of the dual circuit evaluation phase, the view of the adversary generated by the simulator is identically distributed to that in the real world. From super-invertibility property [DN07] of Vandermonde matrices, it follows that the $\mathbf{r}_i$ vectors generated by the parties at the end of this step are random and unknown to the adversary. However, since the adversary is malicious, these vectors may not have the right correlation. Nevertheless, indistinguishability between the view of an adversary in the real protocol

and the transcript generated by the simulator up until the verification step now follows from the privacy of packed secret sharing and the from the fact that the shares sent to the leader are of values that are masked by random values unknown to any party and hence appear completely random to the adversary.

In case no errors are introduced in the protocol before and during the verification step, then the only difference between the real and ideal executions is that the input shares of the honest parties are set to 0 in the simulated transcript. However, by the perfect secrecy of packed secret sharing, this has the same distribution as in a real execution.

In case, some errors were introduced, then the simulator always simulates $f_{\mathsf{checkZero}}$ outputting reject. However, in the real execution, the probability that vector sent to $f_{\mathsf{checkZero}}$ is a non-zero vector is at most $2/|\mathbb{F}|$ and if indeed a non-zero vector is sent to $f_{\mathsf{checkZero}}$, it will get detected except with probability $1/|\mathbb{F}|$. Thus overall, in this case the adversary can avoid detection with probability at most $3/|\mathbb{F}|$. Since this is the only difference between the real execution and the ideal simulation, we have that the statistical difference between these distributions is less than $3/|\mathbb{F}|$.

$\square$

Complexity Calculation for our Maliciously Secure Protocol over Large Fields. For each layer in the protocol, we generate $5 \times$ (width of the layer$/\ell$), where $\ell = n/\epsilon$. We have $t = n\left(\frac{1}{2} - \frac{2}{\epsilon}\right)$. In the malicious setting, $n-t \approx n(\frac{1}{2}+\frac{2}{\epsilon})$ of these packed shares can be computed with $5n^2$ communication. Therefore, overall the total communication required to generate all the randomness is the following:

- Correlated randomness for evaluating the circuit on actual inputs: $\frac{|C|}{\frac{n}{\epsilon}\times n(\frac{1}{2}+\frac{2}{\epsilon})}5n^2 = \frac{10\epsilon^2|C|}{\epsilon+4}$.
- Correlated randomness for evaluating the circuit on randomized inputs: $\frac{10\epsilon^2|C|}{\epsilon+4}$.
- Shares of random $\boldsymbol{\alpha}$ vectors: $\frac{2\epsilon|C|(3\epsilon-4)}{\epsilon+4}$

Additional communication required for dual execution of the circuit is $2{\cdot}5{\cdot}n$(width of the layer$/\ell$). Therefore, overall the total communication to generate correlated randomness and for the dual evaluate the circuit is $\frac{(26\epsilon^2-8\epsilon)|C|}{\epsilon+4} + 10|C|\epsilon = \frac{36\epsilon^2|C|+32\epsilon|C|}{\epsilon+4}$. An additional overhead to generate packed

input shares for all inputs is $n^2|\mathcal{I}|$, where $|\mathcal{I}|$ is the number of inputs to the protocol. The communication required to generate shares of randomized inputs is $n^2|\mathcal{I}|$. Finally, the verfication step only requires $2n^2$ communication. Therefore, the total communication complexity is $\frac{36\epsilon^2|C|+32\epsilon|C|}{\epsilon+4}+2n^2|\mathcal{I}|$. As before, the computation complexity will be $O(\log n)$ times the communication complexity.

3.9 Implementation and Evaluation

In this section, we present the details of our implementation and do a detailed comparison with prior work.

3.9.1 Comparison

We start by comparing the concrete efficiency of our protocol based on the calculations from Section 3.7.5, where we show that the total communication complexity of our maliciously secure protocol is $\frac{36\epsilon^2|C|+32\epsilon|C|}{\epsilon+4}+2n^2|\mathcal{I}|$. Recall that our protocol achieves security against $t < n\left(\frac{1}{2}-\frac{2}{\epsilon}\right)$ corruptions; we do our comparison with the state-of-the-art using the same corruption threshold as they consider.

The state-of-the-art in this regime is the $O(n|C|)$ protocol of [FL19] for $t < n/3$ corruptions, that requires each party to communicate approximately $4\frac{2}{3}$ field elements per multiplication gate. In contrast, for $n = 125$ parties and $t < n/3$ corruptions, our protocol requires each party to send approximately $2\frac{3}{4}$ field elements per gate, in expectation. Notice that while we require parties to communicate for every gate in the circuit, [FL19] only requires communication per multiplication gate. However, it is easy to see that for circuits with approximately 65% multiplication gates, our protocol is expected (in theory) to outperform [FL19] for 125 parties.

As discussed earlier, a nice advantage of $O(|C|)$ protocols is that the per-party communication in these protocols goes down as the number of parties increases. For instance, for the same corruption threshold of $t < n/3$, and $n = 150$ parties, our protocol would (on paper) only require each party to communicate $2\frac{1}{3}$ field elements per gate. In this case, our protocol is already expected to perform better than [FL19] for circuits that have more that 55% multiplication gates. In fact, as the number

of parties increase, the percentage of the circuit that must be multiplication gates in order to show improvements reduces.

Since the communication complexity of our protocol is dependent on the tunable parameter ϵ (that is directly proportional to the corruption threshold t), the efficiency of our protocol is expected to increase further even for fewer parties, if we allow for lower corruption threshold. For instance, for $t < n/4$ corruptions and $n = 100$ parties, we require per-party communication of $2\frac{1}{7}$ field elements per gate.

Finally, we remark that, the above is only a theoretical comparison. Indeed the complexity calculation in Section 3.7.5 was done assuming the "best-case scenario", e.g., where the circuit is such that it has exactly $n-t$ repetitions of the same kinds of blocks, and that each block has an exact multiple of n/ϵ gates and n is exactly divisible by ϵ etc. In practice, this may not be the case. When the circuit is not perfectly divisible, there may be some "waste," meaning either more randomness will be generated than is needed or some packed secret sharings will not be completely filled. The effects of this waste can be seen in our implementation below, where the runtime may even decrease slightly (see $t = n/3$ for 70 and 90 parties) as the number of parties increases because the division is more efficient.

To make our comparison more concrete, we implement our protocol and evaluate it on different network settings. While we do not get the exact same improvements as derived above (likely due to waste), we clearly demonstrate that our protocol is practical for even small numbers of parties, and becomes more efficient than state-of-the-art for large numbers of parties.

3.9.2 Implementation

We implemented our maliciously secure protocol from Section 3.7.5. Our implementation is in `C++` and uses libscapi [Cry19] to provide communication and circuit parsing. Since this library does not support packed secret sharing or the non-interactive packing of traditional secret shares (Section 3.4.3), we implement them within the context of the library. We note that we do not implement packed secret sharing using FFT and hence the computation time incurred for packed secret sharing a

Table 3.2: Comparing the runtime of our order-C protocol and that of related work. All times are in milliseconds.

Paper	This Work	This Work	[CGH+18]	[FL19]	This Work	This Work	[CGH+18]	[CGH+18]	[CGH+18]
Network Config	LAN				WAN				
t	$n/4$	$n/3$	$n/2$	$n/3$	$n/4$	$n/3$	$n/2$	$n/2$	$n/2$
Depth	1,000	1,000	1,000	20	1,000	1,000	20	100	1,000
$n = 30$	28,709	29,014	12,144	1,241	261,029	298,520	87,355	34,860	376,464*
$n = 50$	36,544	40,537	26,310	1,891	206,475	285,074	128,366	197,321	815,610*
$n = 70$	48,729	54,692	33,294	2,585	186,535	214,575	164,145*	251,286*	1,032,114*
$n = 90$	52,563	54,477	48,927	3,689	278,038	260,995	204,166*	355,167*	1,516,737*
$n = 110$	60,281	62,871	79,728	$> 3,999$	270,558	305,071	256,711*	478,361*	2,471,568*
$n = 150$	-	-	-	-	282,381	315,182	-	-	-
$n = 200$	-	-	-	-	262,621	279,111	-	-	-
$n = 250$	-	-	-	-	301,555	320,477	-	-	-
$n = 300$	-	-	-	-	335,588	378,262	-	-	-

single vector is $O(n^2)$ as opposed to $O(n \log n)$ and the overall total computation of this implemented protocol is $O(n|C|)$. All prior works that we compare to in this section also use an unoptimized $O(n^2)$ implementation of the underlying secret sharing scheme and the total computation complexity of those implementations is $O(n^2|C|)$. We leave an optimized implementation of the packed secret sharing scheme for future work.

Our protocol implementation automatically generates batches of correlated randomness on the fly as needed. During circuit evaluation, gates within each block are divided into packs according to the number of players and the packing constant. Randomness is then retrieved from a pool; if no suitable randomness is available from a previous execution of the randomness generation subprotocol, the players pause to generate a fresh batch of randomness and verify that it is correct. This reduces the need to set aside large amounts of memory at the beginning of computation.

To evaluate our implementation, we generate layered circuits that satisfy the highly repetitive structural requirements. Specifically, we generate a fixed number of layers of constant depth, each containing addition and multiplication gates that are randomly wired together. These layers are then repeated as a group some fixed number of times. Benchmarking on random circuits is common, accepted practice for honest majority protocols [CGH+18, FL19]. We modify the circuit format from a standard one used by libscapi [Cry19] to help make this representation more succinct. Specifi-

cally, because our protocol only operates over layered circuits, we have gates take in wires indexed relatively from the previous layer, instead of using global indices. Additionally, because layers are repeated many times, we just indicate the order of layers, rather than writing out the layers explicitly.

We ran tests in two network deployments, the first to measure the performance independent of network delay and the second to measure the effect of network communication. In our first deployment, we ran all of the parties on a single, large server with two Intel(R) Xeon(R) CPU E5-2695 @ 2.10 GHz. In our second deployment, parties were split evenly across three different AWS regions: us-east-1, us-east-2, and us-west-2. Each party was a separate c4.xlarge instance with 7.5 GB of RAM and a 2.9 GHz Intel Xeon E5-2666 v3 Processor.

We compare our work to the most efficient $O(n|C|)$ work, as there is no comparable work which has been run for a large number of parties.[5] These works only run their protocol for up to 110 parties. Therefore our emphasis is not on direct time result comparisons, but instead on relative efficiency even with small numbers of players. We note that the protocols against which we compare do not require highly repetitive circuits; while this might make it seem like we are performing an apple-to-oranges comparison, there is no efficiency gain for running these $O(n|C|)$ protocols over highly repetitive circuits. As such, the timing results that these works present would hold true for highly repetitive circuits as well, and our comparison is apples-to-apples for highly repetitive circuits (up to the percentage of addition gates, which we discussed above.)

We compare the runtime of our protocol in both our LAN deployment and WAN deployment to [CGH+18, FL19] in Table 3.2 (results are reported for the average protocol execution time over five randomized circuits each with 1,000,000 gates for WAN. LAN results are for a single execution with 1,000,000 gates. Asterisk denote estimated runtimes where data was missing.). Because of differences between our protocol and intended applications, there are several important things to note in this comparison. First, we run all our tests on circuits with depth 1,000 to ensure there is sufficient repetition in the circuit. Furukawa et al. use only a depth 20 circuit in their LAN tests, meaning more parallelism can be leveraged. We note that when Chida et al. increase the

[5] The only protocol to be run on large numbers of parties rests on incomparable assumptions like CRS [WJS+19].

depth of their circuits from 20 to 1,000 in their LAN deployment, the runtime for large numbers of parties increases 5-10x [CGH+18]. If we assume [FL19] will act similarly, we see that their runtime is approximately half of ours, when run with small number of parties. This is consistent with their finding that their protocol is about twice as fast as [CGH+18]. We emphasise that for larger numbers of parties our protocol is expected to perform better.

Because Chida et al. only run their protocol for up to 30 players and up to circuit depth 100 in their WAN deployment, there is missing data for our comparison. We note that their WAN runtimes are consistently just over 30x higher than their LAN deployment. Using this observation, we extrapolate estimated runtimes for their protocol under different configurations, denoted with an asterisk. We emphasise that this estimation is rough, and all these measurements should be interpreted with a degree of skepticism; we include them only to attempt a more consistent comparison to illustrate the general trends of our preliminary implementation.

Our results show that our protocol, even using an un-optimized implementation, is comparable to these works for small numbers of parties (see Table 3.2). For larger numbers of parties (see Table 3.2), where we have no comparable results, there is an upward trend in protocol execution time. This could be a result of networking overhead or varying levels of network congestion when each of the experiments was performed. For example, when executing with 250 parties and a corruption threshold of n/4 the difference between the fastest and slowest execution time was over 60,000 ms, whereas in other deployments the difference is as low as 1,000 ms. In general, an increase is also expected as asymptotic complexity has an additive quadratic dependency on n with the input size of the circuit. Overall our experiments demonstrate that our protocol does not introduce an impractical overhead in its effort to achieve $O(|C|)$ MPC. *As the number of parties continues to grow (*e.g. *hundreds or thousands), the benefits of our protocol will become even more apparent*.

Chapter 4

Secure Multiparty Computation with Dynamic Participants

4.1 Technical Overview

We start by briefly discussing some specifics of the model in which we will present our construction. A detailed formal description of our model is provided in Section 4.2.

As discussed earlier, we consider a client-server model where computation proceeds in three phases – input stage, execution stage and output stage (see Figure 1.1). The execution stage proceeds in epochs, where different committees of servers perform the computation. Each epoch ℓ is further divided into two phases: (1) *computation phase*, where the servers in the committee (denoted as $\mathcal{S}^\ell$) perform computation, and (2) *hand-off* phase, where the servers in $\mathcal{S}^\ell$ transfer their states to the incoming committee $\mathcal{S}^{\ell+1}$. Because our goal is to divide the work of the protocol, we impose the efficiency requirement that the computation and communication of each epoch is independent of the depth of the circuit. In order to facilitate the smooth transition of protocol state, we require that at the start of the hand-off phase of epoch ℓ, everyone is aware of committee $\mathcal{S}^{\ell+1}$. We consider security in the presence of an adversary who can corrupt a minority of servers in every committee.

For the remainder of the technical overview, we describe our ideas for the simplified case where

all the committees are disjoint and the size of the committees remain the same across all epochs, denoted as n. Neither of these restrictions are necessary for our protocols, and we refer the reader to the technical sections for further details.

4.1.1 Main Challenges

Designing protocols that are well suited to the fluid MPC setting requires overcoming challenges that are not standard in the static setting. While some of these challenges have been considered previously in isolation in other contexts, the main difficulty is in addressing them at the same time.

1. **Fluidity.** The primary focus of our work is the fluidity of protocols, a measure of how long the servers must remain online in order to contribute to the computation. The fluidity of a protocol is the number of rounds of interaction in a single epoch, and we say that a protocol achieves maximal fluidity if there is only a *single* round in each epoch. Designing protocols with maximal fluidity means that the computation phase of an epoch must be "silent" (i.e., non-interactive), and the hand-off phase must complete in a single round.

2. **Small State Complexity.** In many classical MPC protocols, the private state held by each party is quite large, often proportional to the size of the circuit (see, e.g. [DN07]). We refer to this as the *state complexity* of the protocol. While state complexity is generally not considered an important measure of a protocol's efficiency, in the fluid MPC setting it takes on new importance. Because the state held by the servers must be transferred between epochs, the state complexity of a protocol contributes directly to its communication complexity. Protocols with large state complexity, say proportional to the size of the circuit, would require each committee to perform a large amount of work, violating our efficiency requirements. Therefore, special attention must be paid to minimize the state complexity of the protocol in the fluid MPC setting.

3. **Secure State Transfer.** As mentioned earlier, we consider adversaries that can corrupt a minority of servers in every committee. As such, state cannot be naively handed off between

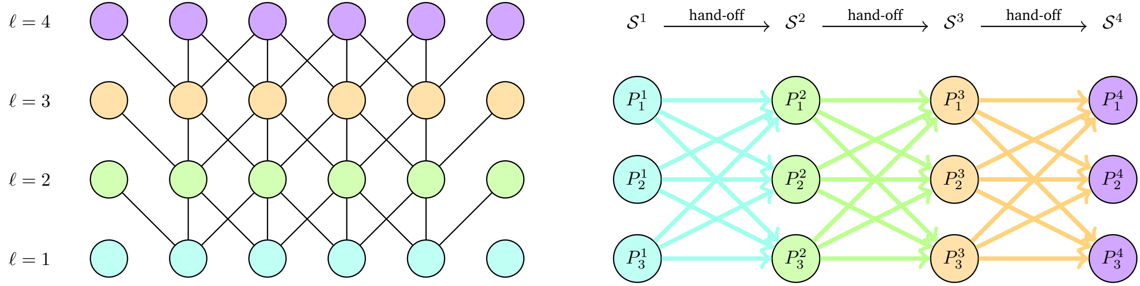


Figure 4.1: **Left:** Part of the circuit partitioned into different layers, indicated by the different colors. **Right:** A visual representation of the flow of information during the modified version of BGW presented in Section 4.1.2.

committees in a one-to-one manner. To illustrate why this is true, consider secret sharing based protocols where the players collectively hold a t-out-of-n secret sharing of the wire values and iteratively compute on these shares. If states were transferred by having each server in committee $\mathcal{S}^i$ choose a unique server in $\mathcal{S}^{i+1}$ (as noted, we assume for convenience that $|\mathcal{S}^i| = |\mathcal{S}^{i+1}|$) and simply sending that new server their state, the adversary would see $2t$ shares of the transferred state, t shares from $\mathcal{S}^i$ and another t shares from $\mathcal{S}^{i+1}$, thus breaking the privacy of the protocol. Fluid MPC protocols must therefore incorporate mechanisms to securely transfer the protocol state between committees.

In this work, we focus our attention on protocols that achieve maximal fluidity. Designing such protocols requires careful balancing between these three factors. In particular, the need for small state complexity makes it difficult to use many of the efficient MPC techniques known in the literature, as we will discuss in more detail below.

4.1.2 Adapting Optimized Semi-honest BGW [GRR98] to Fluid MPC

Despite the challenges involved in the design of fluid MPC protocols, we observe that the protocol by Gennaro et al. [GRR98], which is an optimized version of the semi-honest BGW [BGW88] protocol can be adapted to the fluid MPC setting in a surprisingly simple manner.

Recall that in [GRR98], the parties collectively compute over an arithmetic circuit representation of the functionality that they wish to compute, using Shamir's secret sharing scheme. For each

intermediate wire in the circuit, the following invariant is maintained: the shares held by the parties correspond to a t-of-n secret sharing of the value induced by the inputs on that wire. Evaluating addition gates requires the parties to simply add their shares of the incoming wires, leveraging the linearity of the secret sharing scheme. For evaluating multiplication gates, the parties first locally multiply their shares of the incoming wires, resulting in a distributed degree $2t$ polynomial encoding of the value induced on the output wire of the gate. Then, each party computes a fresh t-out-of-n sharing of this degree $2t$ share and sends one of these *share-of-share* to every other party. Finally, the parties locally interpolate these received shares and as a result, all the parties hold a t-out-of-n sharing of the product. Thus, every multiplication gate requires only one round of communication.

We observe that adapting this version of semi-honest BGW to fluid MPC setting, which we will refer to as Fluid-BGW, is straightforward. The key observation is that the degree reduction procedure of this protocol *simultaneously* re-randomizes the state, so that only a *single round of communication* is required to accomplish both goals. In each epoch, the servers will evaluate all the gates in a *single layer* of the circuit, which may contain both addition and multiplication gates (see Figure 4.1). More specifically, for each epoch ℓ:

Computation Phase: The servers in $\mathcal{S}^\ell$ interpolate the shares-of-shares (received from the previous committee) to obtain a degree t sharing for full intermediary state (for each gate in that layer). Then, they locally evaluate each gate in layer ℓ, possibly increasing the degree of the shares that they hold. Finally, they compute a t-out-of-n secret sharing of the *entire* state they hold, including multiplied shares, added shares and any "old" values that may be needed later in the circuit.

Hand-off Phase: The servers in $\mathcal{S}^\ell$ then send one share of each sharing to each active server in $\mathcal{S}^{\ell+1}$.

The computation phase is non-interactive and the hand-off phase consists of only a single round of communication, and therefore the above protocol achieves maximal fluidity.

Recall that we consider adversaries who can corrupt a minority of t servers in *each committee*, a

significant departure from the classical setting in which a *total* of t parties can be corrupted. At first glance, it may seem as though the adversary can gain significant advantage by corrupting (say) the first t parties in committee $\mathcal{S}^\ell$ and the last t parties in committee $\mathcal{S}^{\ell+1}$. However, since computing shares-of-shares essentially re-randomizes the state, at the end of the hand-off phase of epoch ℓ, the adversary is aware of the (1) nt shares-of-shares that were sent to the last t corrupt servers during the hand-off phase of epoch ℓ and (2) $(n-t) \times t$ shares-of-shares that the first t corrupt servers in $\mathcal{S}^\ell$ sent to the $(n-t)$ honest servers in $\mathcal{S}^{\ell+1}$. This is in fact no different than regular BGW. Since the partial information that the adversary has about the states of the $(n-t)$ honest servers in $\mathcal{S}^{\ell+1}$ only corresponds to t shares of their individual states, privacy is ensured.

4.1.3 Compiler for Malicious Security

Having established the feasibility of semi-honest MPC with maximal fluidity, we now describe our ideas for transforming semi-honest fluid MPC protocols into ones that achieve security against malicious adversaries. Our goal is to achieve two salient properties: (1) *fluidity preservation*, i.e., preserve the fluidity of the underlying protocol, (2) *multiplicative overhead of 2* in the complexity of the underlying protocol.

Shortcomings of Natural Solutions. Consider a natural way of achieving malicious security: after each gate evaluation, the servers perform a check that the gate was properly evaluated, as is done in the malicious-secure version of BGW [BGW88]. However, known techniques for implementing gate-by-gate checks rely on primitives such as verifiable secret sharing (among others) that require *additional interaction* between the parties. Such a strategy is therefore incompatible with our goal of achieving maximal fluidity, which requires a single round hand-off phase.

Starting Idea: Consolidated Checks. Since performing gate-by-gate checks is not well-suited to fluid MPC, we consider a *consolidated check* approach to malicious security, where the correctness of the computation (of the entire circuit) is checked *once*. This approach has previously been studied in the design of efficient MPC protocols [DPSZ12, GIP+14, GIP15, NV18, CGH+18, FL19, GSZ20].

In this line of work, [GIP+14] made an important observation, that *linear-based MPC protocols* (a natural class of semi-honest honest-majority MPC protocols) are secure *up to additive attacks*, meaning any strategy followed by a malicious adversary is equivalent to injecting an additive error on each wire in the circuit. They use this observation to first compile the circuit into another circuit that automatically detects errors, e.g., AMD circuits and then run a semi-honest protocol on this modified circuit to get malicious security. Many other works [GIP15, GIW16] follow suit.

Assuming that the same observation caries over to the fluid MPC setting, for feasibility, one could consider running a semi-honest, maximally fluid MPC protocol on such transformed circuits. However, transforming a circuit into an AMD circuit incurs very high overhead in practice. In order to design a more efficient compiler that only incurs an overhead of 2, we turn towards the approach taken by some of the more recent malicious security compilers [NV18, CGH+18, FL19, GSZ20]. In some sense, the ideas used in these works can be viewed as a more efficient implementation of the same idea as above (without using AMD circuits).

Roughly speaking, in the approach taken by these recent compilers, for every shared wire value z in the circuit, the parties also compute a secret sharing of a MAC on z. At the end of the protocol, the parties verify validity of all the MACs in one shot. Given the observation from [GIP+14], it is easy to see that the parties can generate a single, secret MAC key r at the beginning of the protocol and compute $MAC(r, z) = rz$ for each wire z in the circuit. It holds that if the adversary injects an additive error δ on the wire value z, to surpass the check, they must inject a corresponding additive error of $\hat{\delta} = r\delta$ on the MAC. Because r is uniformly distributed and unknown to all servers, this can only happen with probability negligible in the field size. While previously, this approach has primarily been used for improving the efficiency of MPC protocols, we use it in this work for also maximizing fluidity.

Verifying the MACs requires revealing the key r, but this is only done at the *end* of the protocol, as revealing r too early would allow the adversary to forge MACs. Furthermore, to facilitate efficient MAC verification, the parties finish the protocol with the following “condensed” check: they generate random coefficients α_k and use them to compute linear combinations of the wire values and MACs

as follows:

$$u = \sum_{k \in [|C|]} \alpha_k \cdot z_k \text{ and } v = \sum_{k \in [|C|]} \alpha_k \cdot rz_k.$$

Finally, they reconstruct the key r and interactively verify if $v = ru$, before revealing the output shares.

To build on this approach, we first need to show that *linear-based fluid MPC protocols* are also secure up to additive attacks against malicious adversaries. We prove this to be true in Section 4.5 and show that the semi-honest Fluid-BGW satisfies the structural requirement of linear-based fluid MPC protocols. At first glance, it would appear that we can then directly implement the above mechanism to the fluid MPC setting as follows: in the output stage, parties interactively generate shares of α_k, locally compute this linear combination, reconstruct r, and perform the equality check.

To see where this approach falls short, consider the state complexity of this protocol. To perform the consolidated check, parties in the output stage require shares of *all wires in the circuit*, namely z_k and rz_k for $k \in [|C|]$, which must have been passed along as part of the state between each consecutive pair of committees. This means that the state complexity of the protocol is proportional to the *size of the circuit*, which (as discussed earlier) would undermine the advantages of the fluid MPC model. More concretely, this approach would incur at least $|C|$ multiplicative overhead in the communication of the underlying protocol – far higher than our goal of achieving constant overhead.

Incrementally Computing Linear Combination. In order to implement the above consolidated check approach in the fluid MPC setting, we require a method for computing the aforementioned aggregated values that does not require access to the entire intermediate computation during the output stage. Towards this, we observe that the servers can *incrementally* compute u and v throughout the protocol. This can be done by having each committee incorporate the part of u and v corresponding to the *gates evaluated by the previous committee* into the partial sum. That is, committee $\mathcal{S}^\ell$ is responsible for (1) evaluating the gates on layer ℓ, (2) computing the MACs for gates on layer ℓ, and (3) computing the partial linear combination for all the gates *before* layer $\ell - 1$.

Let the output of the kth gate on the ith layer of the circuit be denoted as z_k^i. Apart from the

shares of $z_k^{\ell-1}$ and $rz_k^{\ell-1}$ (for $k \in [w]$), the servers computing layer ℓ of the circuit $\mathcal{S}^\ell$ also receive shares of

$$u_{\ell-2} = \sum_{i \leq \ell-2} \sum_{k \in [w]} \alpha_k^i \cdot z_k^i \text{ and } v_{\ell-2} = \sum_{i \leq \ell-2} \sum_{k \in [w]} \alpha_k^i \cdot rz_k^i$$

from $\mathcal{S}^{\ell-1}$ during hand-off, where α_k^i is a random value associated with the gate outputting z_k^i. While $u_{\ell-2}$ and $v_{\ell-2}$ represent the consolidated check for all gates in the circuit *before* layer $\ell - 1$. $\mathcal{S}^\ell$ then computes shares of

$$u_{\ell-1} = u_{\ell-2} + \sum_{k \in [w]} \alpha_k^{\ell-1} \cdot z_k^{\ell-1} \text{ and } v_{\ell-1} = v_{\ell-2} + \sum_{k \in [w]} \alpha_k^{\ell-1} \cdot rz_k^{\ell-1}$$

in addition to shares of the outputs of gates on layer ℓ (z_k^ℓ and rz_k^ℓ) and transfer $u_{\ell-1}$ and $v_{\ell-1}$ to $\mathcal{S}^{\ell+1}$ during hand-off. Note that the final $u = u_d$ and $v = v_d$, where d is the depth of the circuit. This leaves the following main question: how do the servers agree upon the values of α_k^ℓ?

Notice that $|\{\alpha_k^\ell\}_{k \in [w], \ell \in [d]}| = |C|$, therefore generating shares of all the α_k^ℓ values at the beginning of the protocol and passing them forward will, again, yield a protocol that has an excessively large state complexity. Another natural solution might be to have the servers generate α_k^ℓ as and when they need them. However, because our goal is to maintain maximal fluidity, the servers in $\mathcal{S}^j$ for some fixed j cannot generate α_k^j, as this would require communication *within* $\mathcal{S}^j$.

Instead, consider a protocol in which the servers in $\mathcal{S}^{j-1}$ do the work of generating the shares of α_k^j. Each server in $\mathcal{S}^{j-1}$ generates a random value and shares it, sending one share to each server in $\mathcal{S}^j$. The servers in $\mathcal{S}^j$ then combine these shares using a Vandermonde matrix to get correct shares of α_k^j, as suggested by [BTH06]. While this approach achieves maximal fluidity and requires a small state complexity, it incurs a multiplicative overhead of n in the complexity of the underlying semi-honest protocol.[1]

Efficient Compiler. We now describe our ideas for achieving multiplicative overhead of only 2 (for circuits over large fields). In our compiler, we use the above intuition, having each committee,

[1]In the static setting, this technique allows for batched randomness generation, by generating $O(n)$ sharings with $O(n^2)$ messages. In the fluid MPC setting, however, the number of servers *cannot* be known in advance and may not correspond to the width of the circuit. Therefore, such amortization techniques are not applicable.

evaluate gates for its layer, compute MACs for the previous layer, and incrementally add to the sum. In the input stage, the clients generate a sharing of a secret random MAC key r, and secret random values $\beta, \alpha_1, \dots, \alpha_w$. Over the course of the protocol, the servers will incrementally compute values

$$u = \sum_{\ell \in [d]} \sum_{k \in [w]} (\alpha_k (\beta)^\ell) \cdot z_k^\ell \text{ and } v = \sum_{\ell \in [d]} \sum_{k \in [w]} (\alpha_k (\beta)^\ell) \cdot r z_k^\ell$$

where z_k^ℓ is the output of the kth gate on level ℓ, $(\beta)^\ell$ is β raised to the ℓth power, and $\alpha_k(\beta)^\ell$ is the "random" coefficient associated with it. At the end of the protocol, the parties verify whether $v = ru$.

Notice that at the beginning of the execution stage, the servers do not have shares of $(\alpha_k(\beta)^\ell)$ for $\ell > 0$, but they have the necessary information to compute a valid sharing of this coefficient in parallel with the normal computation, namely $\beta, \alpha_1, \dots, \alpha_w$. To compute the coefficients, we require that the servers computing layer ℓ are given shares of $(\alpha_k(\beta)^{\ell-1})$ and β by the previous set of servers, in addition to the shares of the actual wire values. The servers in $\mathcal{S}^\ell$ then use these shares to compute shares of (1) the values z_k^ℓ on outgoing wires from the gates on layer ℓ, (2) the partial sums by adding the values computed in the previous layer $u_{\ell-1} = u_{\ell-2} + (\alpha_k(\beta)^{\ell-1}) \cdot z_k^{\ell-1}$ and $v_{\ell-1} = v_{\ell-2} + (\alpha_k(\beta)^{\ell-1}) \cdot r z_k^{\ell-1}$, and (3) the coefficients for the next layer $(\alpha_k(\beta)^\ell) = \beta \cdot \alpha_k(\beta)^{\ell-1}$. All of this information can be securely transferred to the next committee.

We give a simplified sketch to illustrate why this check is sufficient. Let $\epsilon_{z,k}^\ell$ (and $\epsilon_{rz,k}^\ell$ resp.) be the additive error introduced by the adversary on the computation of z_k^ℓ ($r z_k^\ell$ resp.).

As before, the check succeeds if

$$r \cdot \sum_{\ell \in [d]} \sum_{k \in [w]} (\alpha_k(\beta)^\ell)(z_k^\ell + \epsilon_{z,k}^\ell) = \sum_{\ell \in [d]} \sum_{k \in [w]} (\alpha_k(\beta)^\ell)(r z_k^\ell + \epsilon_{rz,k}^\ell)$$

Let the qth gate on level m be the first gate where the adversary injects errors $\epsilon_{z,q}^m$ and $\epsilon_{rz,q}^m$. The above equality can be re-written as.

$$\alpha_q \left[\sum_{\ell=m}^{d} ((\beta)^\ell \epsilon_{rz,q}^\ell) - r \sum_{\ell=m}^{d} ((\beta)^\ell \epsilon_{z,q}^\ell) \right] = r \cdot \sum_{\ell=m}^{d} \sum_{\substack{k \in [w] \\ k \neq q}} (\alpha_k(\beta)^\ell)(z_k^\ell + \epsilon_{z,k}^\ell) - \sum_{\ell=m}^{d} \sum_{\substack{k \in [w] \\ k \neq q}} (\alpha_k(\beta)^\ell)(r z_k^\ell + \epsilon_{rz,k}^\ell)$$

This holds only if either (1) $\sum_{\ell=m}^{d} ((\beta)^\ell \epsilon_{z,q}^\ell) = 0$ and $\sum_{\ell=m}^{d} ((\beta)^\ell \epsilon_{rz,q}^\ell) = 0$. The key point is that

since these are polynomials in β with degree at most d, the probability that β is equal to one of its roots is $d/|\mathbb{F}|$. Or if (2) $r = \sum_{\ell=m}^{d}((\beta)^{\ell}\epsilon_{rz,q}^{\ell})(\sum_{\ell=m}^{d}((\beta)^{\ell}\epsilon_{z,q}^{\ell}))^{-1}$. Since r is uniformly distributed, this happens only with probability $1/|\mathbb{F}|$.

This analysis is significantly simplified for clarity and the full analysis is included in Section 4.6.2 . Note that the adversary can inject additive errors on r and β, since these values are also re-shared between sets of servers. Also, since the α values for the gates on level $\ell > 0$ are computed using a multiplication operation, the adversary can potentially inject additive errors on these values as well. However, we observe that the additive errors on the value of β and consequently on the α values associated with the gates on higher levels, does not hamper the correctness of output. But the errors on the value of r, do need to be taken into consideration. The analysis in the Section 4.6.2 addresses how these errors can be handled, making it non-trivial and notationally complicated, but the core intuition remains the same.

We note that we are not the first to consider generating multiple random values by raising a single random value to consecutively larger powers. In particular, [DPSZ12] performs consolidated checks by taking a linear combination of all wire values, the coefficients for which need to be generated securely, i.e. be randomly distributed and authenticated. But this generation is expensive, so they generate a single secure value and derive all other values by raising it to consecutively larger powers. A consequence of this technique is that once the single secure value is revealed, the exponentiations are done locally and therefore precludes any introduction of errors in this computation for the honest parties. Although this technique might seem similar to ours, our specific implementation is different and for a different purpose, namely, achieving maximal fluidity together with small constant multiplicative overhead.

A roadmap to our constructions can be found in Section 4.4.

4.2 Fluid MPC

In this section, we give a formal treatment of the fluid MPC setting. We start by describing the model of computation and then turn to the task of defining security. Our goals in this section are twofold: first, we illustrate that there are many possible modeling parameters to choose from in the fluid MPC setting. Second, we highlight the modeling choices that we make for the protocols we describe in later sections. Before beginning, we reiterate that the functionalities considered in this setting can be represented by circuits where the depth of such circuits are large.

Model of Computation. We consider a *client-server* model of computation where a set of clients $\mathcal{C}$ want to compute a function over their private inputs. The clients delegate the computation of the function to a set of servers $\mathcal{S}$. Unlike the traditional client-server model [CDI05, DI05, DI06] where every server is required to participate in the entire computation (and hence, remain online for its entire duration), we consider a dynamic model of computation where the servers can volunteer their computational resources for *part of the computation* and then potentially go offline. That is, the set of servers is not fixed in advance.

We adopt terminology from the execution model used in the context of permissionless blockchains [PSs17, PS17, GKL15]. The protocol execution is specified by an interactive Turing Machine (ITM) $\mathcal{E}$ referred to as the *environment*. The environment $\mathcal{E}$ represents everything that is external to the protocol execution. The environment generates inputs to all the parties, reads all the outputs and additionally can interact in an arbitrary manner with an adversary $\mathcal{A}$ during the execution of the protocol.

Protocols in this execution model proceed in rounds, where at the start of each round, the environment $\mathcal{E}$ can specify an input to the parties, and receive an output from the corresponding parties at the end of the round. We also allow the environment $\mathcal{E}$ to spawn new parties at any point during the protocol. The parties have access to point-to-point and broadcast channels. In addition, we assume fully synchronous message channels, where the adversary does not have control over the delivery of messages. This is the commonly considered setting for MPC protocols.

4.2.1 Modeling Dynamic Computation

In a fluid MPC protocol, computation proceeds in three stages:

Input Stage: In this stage, the environment $\mathcal{E}$ hands the input to the clients at the start of the protocol, who then pre-process their inputs and hand them off to the servers for computation.

Execution Stage: This is the main stage of computation where only the servers participate in the computation of the function.

Output Stage: This is the final stage where only the clients participate in order to reconstruct the output of the function. The output is then handed to the environment.

The clients only participate in the input and output stages of the protocol. Consequently, we require that the computational complexity of both the input and the output stages is *independent* of the depth of the functionality (when represented as a circuit) being computed by the protocol. Indeed, a primary goal of this work is to offload the computation work to the servers and a computation-intensive input/output phase would undermine this goal.

We wish to capture dynamism for the bulk of the computation, and thus model dynamism in the *execution stage* of the protocol (rather than the input and output stages). In the following, we highlight the key modeling choices for the protocols we present by displaying them in **bold font in color**.

Epoch. We model the progression of the execution stage in discrete steps referred to as *epochs*. In each epoch ℓ, only a subset of servers $\mathcal{S}^\ell$ participate in the computation. We refer to this set of servers $\mathcal{S}^\ell$ as the **committee** for epoch ℓ. An epoch is further divided into two phases, illustrated in Figure 4.2:

Computation Phase: Every epoch begins with a computation phase where the servers in the committee $\mathcal{S}^\ell$ perform computation over their local states, possibly involving multiple rounds of interaction with each other. We require that the computation and communication complexity of an epoch should be *independent of the depth of the circuit*.

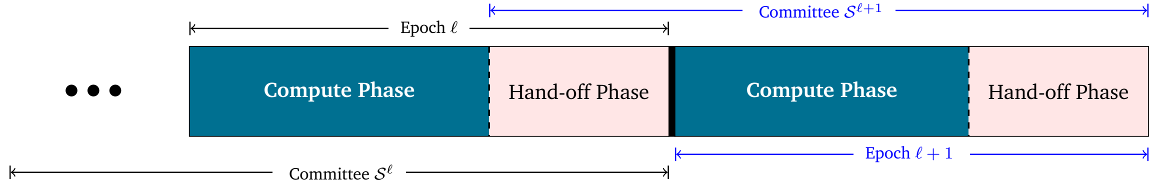


Figure 4.2: Epochs ℓ and $\ell+1$

Hand-off Phase: The epoch then transitions to a hand-off phase where the committee $\mathcal{S}^{\ell}$ transfers the protocol state to the next committee $\mathcal{S}^{\ell+1}$. As with the computation phase, this phase may involve multiple rounds of interaction. When this phase is completed, epoch $\ell+1$ begins.

Fluidity. We define the notion of *fluidity* to measure the minimum commitment that a server needs to make for participating in the execution stage.

Definition 7 (Fluidity). *Fluidity is defined as the number of* rounds *of interaction within an epoch.*

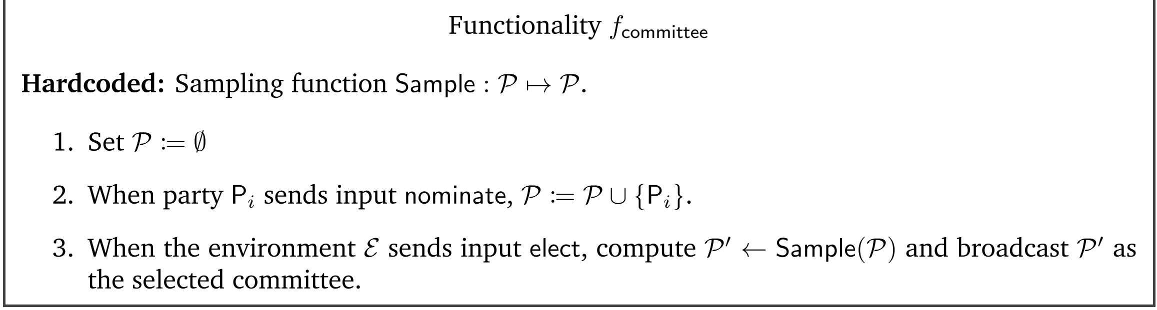
Clearly, the fewer the number rounds in an epoch, the more "fluid" the protocol. We say that a protocol has **maximal fluidity** when the number of rounds in an epoch is 1. We emphasize that this is only possible when the computation phase of an epoch is completely *non-interactive*, i.e., the servers only perform local computation on their states without interacting with each other. This is because the hand-off phase must consist of at least one round of communication. *In this work, we aim to design protocols with **maximal fluidity**.*

4.2.2 Committees

We now explore modeling choices for committees. We address three key aspects of a committee – its formation, size and possible overlap with other committees. Along the way, we also discuss how long a server needs to remain *online*.

Committee Formation. From our above discussion on computation progressing in epochs, we consider two choices for *committee formation*:

Static. In the most restrictive choice, the environment determines right at the start, which



Functionality $f_{\mathsf{committee}}$

Hardcoded: Sampling function $\mathsf{Sample} : \mathcal{P} \mapsto \mathcal{P}$.

1. Set $\mathcal{P} := \emptyset$
2. When party P_i sends input nominate, $\mathcal{P} := \mathcal{P} \cup \{\mathsf{P}_i\}$.
3. When the environment $\mathcal{E}$ sends input elect, compute $\mathcal{P}' \leftarrow \mathsf{Sample}(\mathcal{P})$ and broadcast $\mathcal{P}'$ as the selected committee.

Figure 4.3: Functionality for Committee Formation.

servers will participate in the protocol, *and* the epoch(s) they will be participating in. This in turn determines the committee for every epoch. This means that the servers must commit to their resources ahead of time. We view this choice to be too restrictive and shall not consider it for our model.

On-the-fly. In the other choice, committees are determined dynamically such that **committee for epoch $\ell + 1$ is determined and known to everyone at the start of the hand-off phase of epoch ℓ**. We consider the functionality $f_{\mathsf{committee}}$ described in Figure 4.3 to capture this setting.

In an epoch ℓ, if the environment $\mathcal{E}$ provides input nominate to a party at the start of the round, it relays this message to $f_{\mathsf{committee}}$ to indicate that it wants to be considered in the committee for epoch $\ell+1$. The functionality computes the committee using the sampling function Sample, from the set of parties $\mathcal{P}$ that have been "nominated." The environment $\mathcal{E}$ is also allowed a separate input elect that specifies the cut-off point for the functionality to compute the committee. The cut-off point corresponds to the start of the hand-off phase of epoch ℓ where the parties in $\mathcal{S}^\ell$ are made aware of the committee $\mathcal{S}^{\ell+1}$ via a broadcast from $f_{\mathsf{committee}}$.

We consider two possible committee choices in this dynamic setting below.

Volunteer Committees. One can view the servers as "volunteers" who sign up to participate in the execution stage whenever they have computational resources available. Essentially anyone, who wants to, can join (up until the cut-off point) in aiding with the computation. This can be implemented by simply setting the sampling function Sample in $f_{\mathsf{committee}}$ to be the identity function, i.e. a party is included in the committee for epoch $\ell + 1$ if and only if it

sent a nominate to $f_{\mathsf{committee}}$ during the computation phase of epoch ℓ.

Elected Committees. One could envision other sampling functions that implement a *selection* process using a participation criterion such as the cryptographic sortition [GHM+17a] considered in the context of proof of stake blockchains. The work of [BGG+20] considers the function Sample that is additionally parameterized by a probability p; for each party in $\mathcal{P}$, Sample independently flips a coin that outputs 1 with probability p, and only includes the party in the final committee if the corresponding coin toss results in the value 1. To ensure that all parties are considered in the selection process, one can simply require that every party sends a nominate to $f_{\mathsf{committee}}$ in each epoch. Committee election has also been studied in different network settings; e.g., the recent work of [WJS+19] provides methods for electing committees over TOR [DMS04].

Both of the above choices have direct consequences on the corruption model. The former choice of volunteer committees models protocols that are accessible to anyone who wants to participate. However, an adversary could misuse this accessibility to corrupt a large fraction of (maybe even all) participants of a committee. As such, we view this as an *optimistic* model since achieving security in this model can require placing severe constraints on the global corruption threshold.

The latter choice of elected committees can, by design, be viewed as a semi-closed system since not everyone who "volunteers" their resources are selected to participate in the computation. However, by using an appropriate sampling function, this selection process can potentially ensure that the number of corruptions in each committee are kept within a desired threshold.

We envision that the choice of the specific model (i.e. the sampling function Sample) is best determined by the environment the protocol is to be deployed in and the corruption threshold one is willing to tolerate. (We discuss the latter implication in Section 4.2.3.) Our protocol design is agnostic to this choice and only requires that the committee $\mathcal{S}^{\ell}$ knows committee $\mathcal{S}^{\ell+1}$ at the start of the hand-off phase.

Participant Activity. We say that a server is *active* within an epoch if it either (a) performs some

protocol computation, or (b) sends/receives protocol messages. Clearly, a server P is active during epoch ℓ only if it belongs to $\mathcal{S}^{\ell} \cup \mathcal{S}^{\ell+1}$. When extending this notion to a committee, we say committee $\mathcal{S}^{\ell}$ is active from the beginning of the hand-off phase in epoch $\ell - 1$ to the end of the hand-off phase in epoch ℓ (see Figure 4.2).

We say that a server is "online" if it is active (in the above sense) or simply passively listening to broadcast communication. A protocol may potentially require a server to be online throughout the protocol and keep its local state up-to-date as a function of all the broadcast protocol messages (possibly for participation at a later stage). In such a case, while a server may not be performing active computation throughout the protocol, it would nevertheless have to commit to being present and listening throughout the protocol. To minimize the amount of online time of participants, ideally one would like servers to be online only when active.

Committee Sizes. In view of modeling committee members signing up as and when they have available computational resources, we allow for **variable committee sizes in each epoch**. This simply follows from allowing the environment $\mathcal{E}$ to determine how many parties it provides the nominate input. For simplicity, we describe our protocol in the technical sections for the simplified setting where the committee sizes in each epoch are equal and indicate how it extends to the variable committee size setting. An alternative choice would be to require the committee to have a fixed size, or change sizes at some prescribed rate. These choices might be more reasonable under the requirement that servers announce their committee membership at the start of the protocol.

Committee Overlap. In our envisioned applications, participants with available computational resources will sign up more often to be a part of a committee (see Remark 1). In view of this, we make **no restriction on committee overlap**, i.e., we allow a server to volunteer to be in multiple epoch committees. As we discuss below, this has some bearing on modeling security for the protocol.

Remark 1 (Weighted Computation.). *We note that our model naturally allows for a form of* weighted computation*, where the amount of work performed by a participant is proportional to its available resources. This is because a participant (i.e., a server) can choose to participate in a number of epochs*

proportional to its available resources.

4.2.3 Security

As in traditional MPC, there are various choices for modeling corruption of parties to determine the number of parties that can be corrupted (i.e., honest vs dishonest majority) as well as the time of corruption (i.e., static vs adaptive corruption). The environment $\mathcal{E}$ can determine to corrupt a party, and on doing so, hands the local state of the corrupted party to the adversary $\mathcal{A}$. For a semi-honest (passive) corruption, $\mathcal{A}$ is only able to continue viewing the local state, but for a malicious (active) corruption, $\mathcal{A}$ takes full control of the party and instructs its behavior subsequently.

Corruption Threshold. We consider an *honest-majority* model for fluid MPC where we restrict $(\mathcal{A}, \mathcal{E})$ to the setting where the adversary **$\mathcal{A}$ controls any minority of the clients** as well as **any minority of servers in every committee in an epoch**.

We discuss the impact of the choice of committee formation on corruption threshold:

- **Volunteer Committee.** In the volunteer setting, ensuring honest majority in each epoch may be difficult; as such we view it as an optimistic model. In the extreme case, honest-majority per epoch can be enforced by assuming the global corruption threshold to be $N/2E$ where E is the total number of epochs and N is the total number of parties across all epochs.
- **Elected Committee.** In the elected committee model, the committee selection process may enforce an honest majority amongst the selected participants in every epoch. The work of [BGG+20] enforces this via a cryptographic sortition process in proof-of-stake blockchains where an honest majority of stake is assumed (in fact they require a larger stake fraction to be honest for their committee selection).

An alternative model is where an adversary may control a majority of clients and additionally a majority of servers in one or more epochs. We leave the study of such a model for future work.

Corruption Timing. Given that the protocol progresses in discrete steps, and knowledge of committees may not be known in advance, it is important to model when an adversary can specify the list

of corrupted parties. For clients, this is straightforward: we assume that the environment $\mathcal{E}$ specifies the list of corrupted clients at the start of the protocol, i.e. **we assume static corruption for the clients**. Since the servers perform the bulk of the computation, and their participation is already dynamic, there are various considerations for corruption timing. We consider two main aspects below: *point of corruption* and *effect on prior epochs*.

Point of corruption: When the committee $\mathcal{S}^\ell$ is determined at the start of hand-off phase of epoch $\ell - 1$, the adversary can specify the corrupted servers from $\mathcal{S}^\ell$ in either:

1. a *static* manner, where the environment $\mathcal{E}$ is only allowed to list the set of corrupted servers when the committee $\mathcal{S}^\ell$ is determined; or
2. an *adaptive* manner, where the environment $\mathcal{E}$ can corrupt servers in $\mathcal{S}^\ell$ adaptively up until the end of epoch ℓ, i.e. while they are active.

Effect on prior epochs: We consider the effect of the adversary corrupting parties during epoch ℓ on prior epochs.

1. *No retroactive effect:* In this setting, the corruption of servers during epoch ℓ has no bearing on any epoch $j < \ell$, i.e. the adversary does not learn any additional information about epoch j at epoch ℓ. This model can be achieved in two ways:

   Erasure of states: If servers in $\mathcal{S}^j$ *erase* their respective local states at the end of epoch j, then even if the server were to participate in epoch ℓ (i.e. $\mathcal{S}^j \cap \mathcal{S}^\ell \neq \emptyset$), the adversary would not gain any additional information when the environment $\mathcal{E}$ hands over the local state.

   Disjoint committees: If the sets of servers in each epoch are disjoint, by corrupting servers in epoch ℓ, the adversary cannot learn anything about prior epochs.

   We note that for any protocol that is oblivious to the real identities of the servers (i.e. the protocol doesn't assume any prior state from the servers), the two methods of achieving *no retroactive effect*, i.e. erasures and disjoint committees are equivalent. This follows from the fact that servers do not have to keep state in order to rejoin computation, and therefore

from the point of view of the protocol and for all purposes, are equivalent to new servers.[2]

2. *Retroactive effect:* In this setting, the adversary is allowed limited information from prior epochs. Specifically, when corrupting a server $P \in \mathcal{S}^\ell$ in epoch ℓ, the adversary learns private states of the server in all prior epochs (if the server has been in a committee before). Therefore, the P is then assumed to have been (passively) corrupt in every epoch $j < \ell$. In order to prevent the adversary from arbitrarily learning information about prior epochs, the adversary is limited to corrupting servers in epoch ℓ as long as corrupting a server P and its retroactive effect of considering P to be corrupted in all prior epochs does not cross the corruption threshold in *any* epoch.

One could consider models with various combinations of the aforementioned aspects. We will narrow further discussion to two models of the adversary:

Definition 8 (R-adaptive Adversary). *We say that the $(\mathcal{A}, \mathcal{E})$ results in an R-adaptive adversary $\mathcal{A}$ if the environment $\mathcal{E}$ can statically corrupt a set T of the clients (at the start of the protocol) and corrupt the servers in an adaptive manner with retroactive effect. Specifically, in epoch ℓ, the environment $\mathcal{E}$ can adaptively choose to corrupt a set of servers $T^\ell \subset [n_\ell]$ from the set $\mathcal{S}^\ell$, where T^ℓ corresponds to a canonical mapping based on the ordering of servers in $\mathcal{S}^\ell$. On $\mathcal{E}$ corrupting the server, $\mathcal{A}$ learns its entire past state and can send messages on its behalf in epoch ℓ. The set of servers that $\mathcal{E}$ can corrupt, and its corresponding retroactive effect, will be determined by the corruption threshold τ specifying that $\forall \ell,\ |T^\ell| < \tau \cdot n_\ell$.*

Definition 9 (NR-adaptive Adversary). *We say that the $(\mathcal{A}, \mathcal{E})$ results in an NR-adaptive adversary $\mathcal{A}$ if the environment $\mathcal{E}$ can statically corrupt a set T of the clients (at the start of the protocol) and corrupt the servers in an adaptive manner with* no *retroactive effect. The corruption process is similar to the case of R-adaptive adversaries, except that the environment $\mathcal{E}$ can corrupt any server in epoch ℓ as long as the number of corrupted servers in epoch ℓ are within the corruption threshold. As mentioned earlier, any protocol that achieves security against such an adversary necessarily requires either (a) erasure of*

[2] We would like to point out that if one were to implement point-to-point channels via a PKI, this equivalence may not hold.

state, or (b) disjoint committees.

While our security definition will be general, and encompass both adversarial models, we will consider protocols in the model with **R-adaptive adversary**.

In the above discussions, we have considered corruptions only when servers are *active*. One could also consider a seemingly stronger model where the adversary can corrupt servers when they are *offline*, i.e. no longer *active*. We remark below that our model already captures offline corruption.

Remark 2 (Offline Corruption). *If servers are* offline *once they are no longer* active *i.e. they are not passively listening to protocol messages, then offline corruptions in the retroactive effect model is the same as adaptive corruptions during (and until the end of) the epoch due to the fact that the server's protocol state has not changed since the last time it was active. Going forward, since honest parties do go offline when they are no longer active, we do not specify offline corruptions as they are already captured by our model.*

Remark 3 (Un-corrupting parties). *It might be desirable to consider a model in which a server is initially corrupted by the adversary, but then the adversary eventually decided to "un-corrupt" that server, returning it to honest status. This kind of "mobile adversary" has been studied in some prior works [GHM+17b]. We note that this can be captured in our model by just having the adversary "un-corrupt" a server by making that server leave the computation at the end of the epoch and rely on the natural churn of the network to replace that server.*

Defining Security. We consider a network of m-clients and N-servers $\mathcal{S}$ and denote by $(\vec{n} = (n_1, \ldots, n_E), E)$ the partitioning of the servers into E tuples (corresponding to epochs) where the ℓ-th tuple has n_ℓ parties (corresponding to committee in the ℓ-th epoch), i.e. $\mathcal{S}^\ell \subset \mathcal{S}$ such that $\forall \ell \in [E]$, $|\mathcal{S}^\ell| = n_\ell$.

Similar to the **client-server** setting, defined in [CDI05, DI05, DI06], only the m clients have an input (and receive output), computing a function $f : X_1 \times \cdots \times X_m \rightarrow Y_1 \times \cdots \times Y_m$, where for each $i \in [m]$, X_i and Y_i are the input and output domains of the i-th client.

We provide a definition of fluid MPC that corresponds to the classical security notion in the MPC

literature called **security with abort**, but note that other commonly studied security notions can also be defined in this setting in a straightforward manner. The security of a protocol (with respect to a functionality f) is defined by comparing the real-world execution of the protocol with an ideal-world evaluation of f by a trusted party. More concretely, it is required that for every adversary $\mathcal{A}$, which attacks the real execution of the protocol, there exist an adversary Sim, also referred to as a simulator in the ideal-world such that no environment $\mathcal{E}$ can tell whether it is interacting with $\mathcal{A}$ and parties running the protocol or with Sim and parties interacting with f. As mentioned earlier, the environment $\mathcal{E}$ (i) determines the inputs to the parties running the protocol in each round; (ii) sees the outputs to the protocol; and (iii) interacts in an arbitrary manner with the adversary $\mathcal{A}$. In this context, one can view the environment $\mathcal{E}$ as an interactive distinguisher.

It should be noted that it is only the clients that have inputs to the protocol π. While the servers have no input, the environment $\mathcal{E}$, in any round, can provide it with the input nominate upon which the server relays this message to the ideal functionality to indicate it is volunteering for the committee in the subsequent epoch. These servers have no output, so do not relay any information back to $\mathcal{E}$.

In the **real execution** of the $(\vec{n}, E)$-party protocol π for computing f in the presence of $f_{\mathsf{committee}}$ proceeds first with the environment passing the inputs to all the clients, who then pre-process their inputs and hand it off to the servers in $\mathcal{S}^1$. The protocol then proceeds in epochs as described earlier in the presence of an adversary $\mathcal{A}$ and environment $\mathcal{E}$. $\mathcal{E}$ at the start of the protocol chooses a subset of clients $T \subset [m]$ to corrupt and hands their local states to $\mathcal{A}$. As discussed, the corruption of the clients is static, and thus fixed for the duration of the protocol. The honest parties follow the instructions of π. Depending on whether $\mathcal{A}$ is R-adaptive or NR-adaptive, $\mathcal{E}$ proceeds with adaptively corrupting servers and handing over their states to $\mathcal{A}$ who then sends messages on their behalf.

The execution of the above protocol defines $\mathsf{REAL}_{\pi,\mathcal{A},T,\mathcal{E},f_{\mathsf{committee}}}(z)$, a random variable whose value is determined by the coin tosses of the adversary and the honest players. This random variable contains (a) the output of the adversary (which may be an arbitrary function of its view); (b) the outputs of the uncorrupted clients; and (c) list of all the corrupted servers $\{T^\ell\}_{\ell\in[E]}$.

The **ideal world execution** is defined similarly to prior works. We formally define the ideal execution for the case of retroactive adaptive security, and the analogous definition for non-retroactive adaptive security can be obtained by appropriate modifications. Roughly, in the ideal world execution, the participants have access to a trusted party who computes the desired functionality f. The participants send their inputs to this trusted party who computes the function and returns the output to the participants.

More formally, an ideal world execution for a function f in the presence of $f_{\mathsf{committee}}$ with adversary Sim proceeds as follows:

– **Clients send inputs to the trusted party:** The clients send their inputs to the trusted party, and we let x_i' denote the value sent by client P_i. The adversary Sim sends inputs on behalf of the corrupted clients.

– **Corruption Phase of servers:** The trusted party initializes $\ell = 1$. Until Sim indicates the end of the current phase (see below), the following steps are executed:

1. Trusted party sends ℓ to Sim and initializes an *append-only* list $\mathsf{Corrupt}^{\ell}$ to be $\emptyset$.
2. Sim then sends pairs of the form (j, i) where j denotes epoch number and i denotes the *index of the corrupted server* in epoch $j \leq \ell$. Upon receiving this, the trusted party appends i to the list $\mathsf{Corrupt}^{j}$. This step can be repeated multiple times.
3. Sim sends $\mathsf{continue}$ to the trusted party, and the trusted party increments ℓ by 1.

Sim may also send an abort message to the trusted party in this phase in which case the trusted party sends $\perp$ to all honest clients and stops. Else, Sim sends $\mathsf{next\ phase}$ to the trusted party to indicate the end of the current phase.

The following steps are only executed if the Sim has not already sent an abort message to the trusted.

– **Trusted party sends output to the adversary:** The trusted party computes $f(x_1', \ldots, x_m') = (y_1, \ldots, y_m)$ and sends $\{y_i\}_{i \in T}$ to the adversary Sim.

– **Adversary instructs trust party to abort or continue:** This is formalized by having the adversary send either a continue or abort message to the trusted party. In the latter case, the trusted party sends to each uncorrupted client P_i its output value y_i. In the former case, the trusted party sends the special symbol $\perp$ to each uncorrupted client.

– **Outputs:** Sim outputs an arbitrary function of its view, and the honest parties output the values obtained from the trusted party.

Sim also interacts with the environment $\mathcal{E}$ in an identical manner to the real execution interaction between $\mathcal{E}$ and $\mathcal{A}$. In particular this means, Sim cannot rewind $\mathcal{E}$ or look at its internal state. The above ideal execution defines a random variable $\mathsf{IDEAL}_{\pi,\mathsf{Sim},T,\mathcal{E},f_{\mathsf{committee}}}(z)$ whose value is determined by the coin tosses of the adversary and the honest players. This random variable containing the (a) output of the ideal adversary Sim; (b) output of the honest parties after an ideal execution with the trusted party computing f where Sim has control over the adversary's input to f; and (c) the lists $\left\{\mathsf{Corrupt}^{\ell}\right\}_{\ell}$ of corrupted servers output by the trusted party. If Sim sends abort in the *corruption phase of the server*, the trusted party outputs the lists that have been updated until the point the abort message was received from Sim.

Having described the real and the ideal worlds, we now define security.

Definition 10. *Let $f : X_1 \times \cdots \times X_m \to Y_1 \times \cdots \times Y_m$ be a functionality and let π be a fluid MPC protocol for computing f with m clients, N servers and E epochs. We say that π achieves (τ, μ) retroactive adaptive security (resp. non-retroactive adaptive security) if for every probabilistic adversary $\mathcal{A}$ in the real world there exists a probabilistic simulator* Sim *in the ideal world such that for every probabilistic environment $\mathcal{E}$ if $\mathcal{A}$ is R-adaptive (resp. NR-adaptive) controlling a subset of servers $T^{\ell} \subseteq \mathcal{S}^{\ell}$, $\forall \ell \in [E]$ s.t. $|T^{\ell}| < \tau \cdot n_{\ell}$ and less than $\tau \cdot m$ clients, it holds that for all auxiliary input $z \in \{0,1\}^*$*

$$\mathsf{SD}\left(\mathsf{IDEAL}_{f,\mathsf{Sim},T,\mathcal{E},f_{\mathit{committee}}}(z), \mathsf{REAL}_{\pi,\mathcal{A},T,\mathcal{E},f_{\mathit{committee}}}(z)\right) \leq \mu$$

where $\mathsf{SD}(X, Y)$ is the statistical distance between distributions X and Y.

When μ is a negligible function of some security parameter λ, we say that the protocol π is

τ-secure.

Remark 4. *We note that the above definitions do not explicitly state whether the adversary behaves in (a) a semi-honest manner, where the messages that it sends on behalf of the parties are computed as per protocol specification; or (b) a malicious manner, where it can deviate from the protocol specification. Our intention is to give a general definition independent of the type of adversary. In the subsequent description, we will appropriately prefix the adversary with semi-honest/malicious to indicate the power of the adversary.*

This Work. We summarize the fluid MPC model that we focus on in this work , in the definition below.

Definition 11 (Maximally-Fluid MPC with R-Adaptive Security). *We say that a Fluid MPC protocol π is a* **Maximally-Fluid MPC with R-Adaptive Security** *if it additionally satisfies the following properties:*

- ***Fluidity:*** *It has maximal fluidity.*
- ***Volunteer Based Sign-up Model:*** *Committee for epoch $\ell+1$ is determined and known to everyone at the start of the hand-off phase of epoch ℓ where the sampling function for $f_{committee}$ is the identity function. Each epoch can have variable committee sizes, and the committees themselves can arbitrarily overlap. A server is only required to be online during epochs where it is active.*
- ***Malicious R-Adaptive Security:*** *It achieves security as per Definition 10 against malicious R-adaptive adversaries who control any minority ($\tau < 1/2$) of clients and any minority of servers in every committee in an epoch.*

As we have just shown, there are many interesting, reasonable modeling choices that can be made in the study of fluid MPC. While our specific model name may be heavy-handed, we want to ensure that our modeling choices are clear throughout this work. Additionally, we hope to emphasize that our work is an initial foray in the study of fluid MPC and much is to be done to fully understand this setting.

Security Proofs in the Rest of the Paper. While we have presented our model in the fully generalized setting above, moving forward we will find it convenient to avoid notational clutter and make some simplifying assumption to the model for our setting without affecting its security, as we argue below.

1. To start off, we will not prove a composition theorem for our protocols; we simply prove that they are secure in a standalone execution. However, our constructed simulator will work in a straight-line fashion, meaning it will not rewind $\mathcal{E}$; and neither will it require knowledge of $\mathcal{E}$'s internal state.

2. The inputs to the protocol, the client inputs, are determined at the start of the protocol and no other participant has an input. This prevents the environment from adaptively choosing inputs, and we will prove that our protocol holds for *all* choices of client inputs.

3. While the environment can adaptively (adaptive over its view so far) decide to spawn, corrupt or even volunteer parties over the course of the protocol execution, we will prove that our protocol is secure for *any* of the aforementioned operations as long as the corruption threshold is maintained.

4. Lastly, our protocol is oblivious to the choice of the Sample functionality of $f_{\mathsf{committee}}$ as long the parties in epoch ℓ are aware of the committee in epoch $\ell + 1$, $\mathcal{S}^{\ell+1}$, when the hand-off phase of epoch ℓ begins; so we omit calls to $f_{\mathsf{committee}}$. To remove $\mathcal{E}$'s dependence on determining the start of the hand-off phase; we simply assume there is a "cut-off" period within each epoch that starts the hand-off phase. This makes no difference to the security since honest parties simply wait for the hand-off phase to start otherwise.

This in turn means that the environment's view is determined completely by the view of the adversary, and the outputs of the honest parties, i.e. the notion that $\mathcal{E}$ is an interactive distinguisher can be reduced to a non-interactive one. We will therefore find it convenient to denote the ideal world and real world random variables as $\mathsf{IDEAL}_{f,\mathsf{Sim},T}$ and $\mathsf{REAL}_{\pi,\mathcal{A},T}$ respectively.

4.3 Preliminaries

4.3.1 Layered Circuits

We will design a protocol that works for any polynomial-sized arithmetic circuit with a specific structure. In particular, we consider circuits that can be decomposed into well-defined layers such that the output of gates on a layer ℓ are only used as input to the gates on layer $\ell + 1$. We refer to such circuits as *layered circuits*. Apart from the regular addition and multiplication gates, these circuits can additionally have single input *relay gates* that implement the identity operation. We start by giving a formal definition of layered circuits. Later we discuss how any arithmetic circuit can be transformed into a layered circuit.

Definition 12 (Layered Circuits). *An arithmetic circuit C over a field $\mathbb{F}$ with depth d and maximum width w is said to be a* layered circuit*, if it satisfies the following properties:*

– *The circuit C can be decomposed into d distinct and well-defined layers/layers such that the gates on layer $\ell \in [d]$ take only output wires coming from gates on layer $\ell - 1$ as input.*

– *layer $\ell = 0$ is a special layer consisting of special gates called* input gates*. These gates have in-degree 0. In some cases, we also allow these gates with in-degree 0 to be labeled as* random input gates*. As the name suggests, random input gates output random values. The output of gates in this layer act as inputs to the gates on layer $\ell = 1$.*

– *The circuit consists of another special type of gates called* output gates *on layer $\ell = d+1$. These gates have out-degree 0. The output of gates on layer $\ell = d$ are inputs to the output gates.*

– *Apart from the input and output gates, the circuit consists of the following types of gates:*

  – ***Addition Gates:*** *These gates have arbitrary in-degrees and out-degrees. Given inputs $x_1, \ldots, x_q \in \mathbb{F}$ on the respective input wires, addition gates output $\sum_{i=1}^{q} x_i$ on each of their output wires.*

  – ***Addition-by-Constant Gates:*** *These gates have an in-degree of one and arbitrary out-degree. Given input $x \in \mathbb{F}$, addition-by-constant gates output $(x + c)$ on each of their output wires, where $c \in \mathbb{F}$ is some constant hardwired in the gate.*

– ***Multiplication Gates:*** *These gates have in-degree two and arbitrary out-degrees. Given inputs* $x, y \in \mathbb{F}$ *on the respective input wires, multiplication gates output* $x \cdot y$ *on each of their output wires.*

– ***Multiplication-by-Constant Gates:*** *These gates have in-degree one and arbitrary out-degree. Given input* $x \in \mathbb{F}$*, multiplication-by-constant gates output* $c \cdot x$ *on each of their output wires, where* $c \in \mathbb{F}$ *is some constant hardwired in the gate.*

– ***Relay Gates:*** *Relay gates have in-degree one and arbitrary out-degree. These gates essentially implement the identity function. Given input* $x \in \mathbb{F}$*, they output* x *on each of their output wires.*

In the following lemma we show that any arithmetic circuit can be converted into a layered circuit as defined above.

Transforming any arithmetic circuit into a layered circuit. We first discuss how to transform circuit with fan-in 2 and fan-out 2 gates and then later discuss how to tranform general circuits into layered circuits.

1. *Circuits with fan-in 2 and fan-out 2 gates.* Lets assume the circuit is such that each gate has fan-in 2 and fan-out 2 (we know that such circuits are complete): Let w_i be the number of number of gates on layer i. Number of output wires coming out of the first layer is $2w_1$. The number of input wires going into the second layer is $2w_2$. The number of output wires from layer 1 that are not consumed by the second layer = $2w_1 - 2w_2$. Which means, we will need to add $2w_1 - 2w_2$ relay gates on the second layer. Now, we have $(2w_1 - 2w_2)$ output wires coming from these newly added relay gates and $2w_2$ wires coming from the original gates on the second layer. Overall, number of output wires coming out of the second layer = $(2w_1 - 2w_2) + 2w_2 = 2w_1$. Similarly, for the third layer, $2w_3$ of these wires will be used as input wires and for the remaining unused $2w_1 - 2w_3$ wires, we will have add relay gates. This combined with the $2w_3$ output wires from the original gates on layer 3, the total number of output wires coming out from layer 3 are now = $(2w_1 - 2w_3) + 2w_3 = 2w_1$. We can extend this argument to show that the total number of additional relay gates that need to be added to

make this circuit into a layered circuit = $\sum_{i\in[2,d]} 2w_1 - 2w_i$.

2. *General Circuits.* When trying to directly transform a general circuit that has gates with unbounded fan-out, into a layered circuit – it is difficult to come up with a clean expression to predict how many relay gates will need to be added. Hence, its cleaner to think of first transforming such a circuit into one that only has gates with fan out 2 and then applying the above transformation. Even though the transformation to circuits where each gate has fan-out 2 may not be "efficient". But this additional overhead somewhat seems unavoidable. As observed earlier, in the fluid MPC model, since sequential computation is divided amongst different committees, state complexity inevitably translates to communication complexity. Indeed, consider a circuit where the output of a gate from the layer is consumed as input by a gate on the last layer. When locally evaluating such a circuit, one must hold on to the output of that gate from the first layer all the way up until the last layer. As a result, when evaluating inside the fluid MPC model, each committee (starting with the first committee) must forward the output of this gate to its subsequent committee, so that the last committee can evaluate the last layer.

4.4 Roadmap to Our Results

In this work, we construct a Maximally-Fluid MPC with R-Adaptive Security (see Definition 11). In this section, we outline the sequence of steps used for obtaining this result.

1. In Section 4.5, we adapt the additive attack paradigm of [GIP+14] to the fluid MPC setting. In particular, we start by formally defining a class of secret sharing based fluid MPC protocols, called "linear-based fluid MPC protocols". We then focus on "weakly private" linear-based fluid MPC protocols, which are semi-honest protocols that additionally achieve a weak notion of privacy against a malicious R-adaptive (see Definition 8) adversary. We show that such weakly private protocols are also secure against a malicious R-adaptive adversary up to "additive attacks".

2. In Section 4.6, we present a general compiler that can transform any linear based fluid MPC

protocol that is secure against a malicious R-adaptive adversary up to additive attacks, into a protocol that achieves security with abort against a malicious R-adaptive adversary. Our resulting protocol only incurs a constant multiplicative overhead in the communication complexity of the original protocol and also preserves its fluidity.

3. In Section 4.7, we adapt the semi-honest BGW [BGW88] protocol to the fluid MPC setting and show that this protocol is both linear-based and weakly private against a malicious R-adaptive adversary, and achieves maximal fluidity.

By using the result in Section 4.5, we establish that the linear-based weakly private protocol described in Section 4.7 is also secure against a malicious R-adaptive adversary up to additive attacks. Finally, compiling this protocol using the compiler from Section 4.6, we obtain a maximally fluid MPC protocol secure against malicious R-adaptive adversaries. In Section 4.8, we implement and evaluate this protocol in various network settings.

Notations. From this point onwards, unless specified otherwise, we denote a fluid MPC protocol that satisfies all the properties listed in Definition 11 except that it may or may not be maximally fluid as a Fluid MPC with R-Adaptive Security and as a Fluid MPC, if the corruption model is also unspecified.

4.5 Additive Attack Paradigm in Fluid MPC

In this section, we formalize the notion of "linear-based" Fluid MPC protocols. Linear-based protocols are a special class of MPC protocols that rely on threshold secret sharing and satisfy some additional structural properties. This notion was previously studied in [GIP+14], we generalize it to the Fluid MPC[3] setting. We discuss these structural properties in more detail in Section 4.5.1.

We analyze the security of linear-based Fluid MPC protocols against malicious R-adaptive adversaries, w.r.t. two security notions (1) weak privacy and (2) security up to additive attacks. We start

[3]As mentioned in the previous section, we emphasize on the use of a different font for the term Fluid MPC. This is because, we define linear-based protocols for a restricted class of fluid MPC protocols that satisfy all the properties listed in Definition 11, except that they may or may not be maximally fluid and are not restricted to any corruption model. We do not restrict ourselves to any corruption model for this definition since it only captures the structural properties of a protocol.

by recalling these security notions as defined in [GIP+14].

– A protocol is said achieve *weak privacy* against a malicious adversary, if its "truncated" view (i.e., its view excluding the last communication round) in the real execution can be simulated by a simulator in the ideal world, who does not query the trusted functionality on the inputs of the corrupt parties.

– A protocol is said to be secure against a malicious adversary *up to additive attacks*, if any malicious strategy of the adversary in the protocol is equivalent to injecting arbitrary additive values on each intermediate wire of the circuit (representing the functionality that the MPC computes). More importantly, these additive values are independent of the inputs of the honest parties. Intuitively, this means that in such a protocol, the privacy of the honest parties' inputs is ensured, but the correctness of output is not guaranteed.

We consider weak privacy in the presence of malicious R-adaptive adversaries[4] and show that a weakly private linear-based Fluid MPC protocol is secure against a malicious R-adaptive adversary up to additive attacks. This corresponds to adapting the proof from [GIP+14] to the fluid MPC setting. The rest of this section is organized as follows. In Section 4.5.1, we define linear-based Fluid MPC protocols and in Section 4.5.2 we formally define weak privacy and security up to additive attacks and establish the above relation between these notions.

4.5.1 Linear-Based Fluid MPC Protocols

We start by giving an overview of linear-based MPC protocols as defined in [GIP+14] and then discuss how we extend this concept to the Fluid MPC setting. A linear protocol satisfies two main properties[5]:

– **Messages:** Each message exchanged by the parties in a linear protocol is either computed as an arbitrary function of their main inputs or as a linear combination of their incoming messages.

[4]In order to adapt the notion of weak privacy in the Fluid MPC setting, we consider a slightly modified variant of this definition, which we discuss in Section 4.5.2

[5]In [GIP+14], the authors consider two different kinds of inputs in a linear protocol-namely the *main* inputs of the parties and their *auxiliary* inputs. In our setting, it suffices for us to consider a simplified version of their definition, where the parties do not have any auxiliary inputs.

– **Output:** The output of each party in a linear protocol is computed as a linear combination of its incoming messages.

We now describe the structure of a linear-based protocol w.r.t. linear protocols. At a high level, the parties in a linear-based MPC protocol collectively evaluate the circuit (representing the functionality that they wish to compute) in a gate-by-gate manner on the secret shared inputs of all parties. Each of these inputs is secret shared at the beginning of the protocol using a linear protocol and the shares correspond to those of a threshold secret sharing scheme. The parties evaluate each gate on the secret shared values using a linear protocol. The output of the parties in this linear protocol is a secret sharing of values on the outgoing wires of that gate. At the end, each party holds a share of the output, which they then reveal to each other and reconstruct the output.

In the context of Fluid MPC, we define linear protocols w.r.t. two sets of parties, where only the first set has inputs and only the second set gets the output. In addition to satisfying all of the properties discussed earlier, we impose a structural requirement. In the Fluid MPC setting, we require that a linear protocol be divided into three main phases: (1) *computation phase*, where only the parties in the first set communicate within themselves, (2) the *hand-off phase*, where both sets of parties communicate with each other and (3) the *output phase*, where the parties in the second set locally compute their output.

In order to adapt the definition of a linear-based protocol in the Fluid MPC setting, we require the parties to necessarily operate on a *layered circuit* (see Definition 12). Similar to any fluid MPC protocol, a linear-based Fluid MPC is also divided into an input stage, execution stage and an output stage. In the *input stage*, the clients and the servers in the first committee participate in a linear protocol that allows the clients to secret share their inputs with the first committee. In the *execution stage*, each committee is responsible for evaluating one layer of the circuit. For each gate in layer ℓ of the circuit, committee $\mathcal{S}^\ell$ and $\mathcal{S}^{\ell+1}$ engage in a linear protocol, where the servers in $\mathcal{S}^\ell$ evaluate the gate and hand-off the shares of its output to the servers in $\mathcal{S}^{\ell+1}$. The last committee $\mathcal{S}^d$ hands-off the shares of the output gates (gates on the last layer) to the clients. The clients reveal the shares that they receive to all the other clients in the *output stage* and reconstruct the output. As a result,

the number of committees (and hence the number of epochs) in a linear-based Fluid MPC is equal to the depth of the layered circuit.

Next, we formally define a linear and linear-based Fluid MPC protocol.

Definition 13 (Linear Protocol). *An $(n_1 + n_2)$-party protocol Π is said to be a linear protocol, over some finite field $\mathbb{F}$ if Π consists of communication amongst the parties in $[1, n_1]$ (called the computation phase), followed by a hand-off phase, where the parties in $[1, n_1]$ communicate with the parties in $[n_1 + 1, n_1 + n_2]$, followed a non-interactive output phase and has the following properties:*

1. ***Inputs.*** *The input of every party P_i, for $i \in [1, n_1]$, is a vector of field elements. Parties in $[n_1 + 1, n_1 + n_2]$ have no inputs.*
2. ***Messages.*** *Each message in Π is a vector of field elements. We require that every message $\vec{m}$ of Π, sent by the parties belongs to one of the following categories:*
   (a) *$\vec{m}$ is some fixed arbitrary function of P_i's inputs.*
   (b) *every entry m_j of $\vec{m}$ is generated as some fixed linear combination of elements of previous messages received by P_i.*
3. ***Outputs.*** *The output of every party P_i, for $i \in [n_1+1, n_1+n_2]$, is a linear function of its incoming messages. The parties in $[1, n_1]$ do not have any output.*

Remark. A linear protocol is said to have maximal fluidity if there is no communication amongst the parties in $[1, n_1]$ and the handoff phase consists of a single round of communication where the parties in $[1, n_1]$ send messages to the parties in $[n_1 + 1, n_1 + n_2]$.

As observed in [GIP+14], the output function can be described as a linear function as follows.

Definition 14 (Output function of a linear protocol). *Let π be a linear protocol for computing a functionality f and let $T \subset [n_1 + 1, n_1 + n_2]$ be a subset of parties. Let $\vec{x}$ be the input to π and let $m_{\mathsf{inp},T}$ be the messages of type 2a in Definition 13 sent by parties in T to themselves during an honest execution of π on $\vec{x}$. In addition, let $\mathsf{m}_{\overline{T}\to T}$ be the messages of type 2b sent by the parties in*

$\overline{T} = [n_1 + 1, n_1 + n_2] \setminus T$ *to the parties in* T *during an honest execution of* π*. We say that a function* out_T *is the output function of* T *in* π *if for any input* $\overrightarrow{x}$ *it holds that*

$$\mathsf{out}_T\left(m_{\mathsf{inp},T}\mathsf{m}_{\overline{T}\to T}\right) = f_T(\overrightarrow{x})$$

where f_T *is the restriction of* f *to the outputs of the parties in* T*.*

The following claim is restated from [GIP+14].

Claim 1. *Let* π *be a linear protocol and let* T *be a set of parties. In addition let* out_T *be the output function of* T *in* π*. Then for any* m_1, m_2, m_1', m_2' *it holds that*

$$\mathsf{out}_T\left(m_1 + m_1', m_2 + m_2'\right) =$$
$$\mathsf{out}_T\left(m_1, m_2\right) + \mathsf{out}_T\left(m_1', m_2'\right)$$

We now define the notion of a *linear based Fluid MPC protocol*. For simplicity, we assume that all clients get the same output.

Parties: The protocol is executed by the following sets of parties:

- *Clients:* $\mathcal{C} := \{P_1, \ldots, P_m\}$
- *Servers:* For each $\ell \in [d]$, $\mathcal{S}^\ell := \{P_1^\ell, \ldots, P_{n_\ell}^\ell\}$, where d is the depth of the circuit representing the functionality that the clients wish to compute. There may or may not be an overlap between these sets of servers.

Definition 15 (Linear-based Fluid MPC protocol). *Let* (share, reconstruct) *be the functions associated with a threshold secret sharing scheme (section 2.2.1). A* n*-client* $\overrightarrow{n}$*-sever Fluid MPC protocol* Π *for computing a single output,* n*-client layered circuit (see Section 4.3.1)* $C : \left(\mathbb{F}^{\mathsf{in}}\right)^n \to \mathbb{F}^{\mathsf{out}}$*, where* t *out of* $m \geq 2t+1$ *clients maybe corrupt,* out *is the output length and* in *is the length of each client's input and where* d *is the depth of* C*, is said to be* linear-based *with respect to the threshold secret sharing scheme if* Π *has the following structure:*

Input Stage. *All the clients $\mathcal{C}$ and the servers $\mathcal{S}^1$ participate in a linear protocol π_{input}, where for every input gate G_i, some client P_j has input x_i. At the end of the protocol, each server in $\mathcal{S}^1$ holds a share for each input gate G_i. Simultaneously, the clients $\mathcal{C}$ and the servers $\mathcal{S}^1$ also participate in a linear protocol π_{rand} for every random input gate G^r_k.*

Execution Stage. *The protocol Π proceeds in stages. In each stage ℓ, all gates in level ℓ of the circuit are evaluated. The gates G^ℓ_k themselves in the level are evaluated in parallel, and at the end of the stage, the servers in $\mathcal{S}^{\ell+1}$ hold a sharing of the output of each G^ℓ_k. For notational convenience we denote by G_c gates of the form G^ℓ_w and by G_a and G_b gates of the form $\mathsf{G}^{\ell-1}_w$. We set $\mathcal{S}^{d+1} = \mathcal{C}$. The evaluation of the gates are done in the following manner*

1. ***addition gate.*** *For every addition gate G^c in C with inputs G^a and G^b, Π evaluates G^c by having the servers in $\mathcal{S}^\ell$ sum its shares corresponding to the outputs of G^a and G^b. The servers in $\mathcal{S}^\ell$ and $\mathcal{S}^{\ell+1}$ then participate in a linear protocol π_{trans} where the inputs of the servers in $\mathcal{S}^\ell$ are the shares computed above.*

2. ***addition by constant gate.*** *For every addition by constant gate G^c in C with inputs G^a and constant b, Π evaluates G^c by having the servers in $\mathcal{S}^\ell$ sum its shares corresponding to the outputs of G^a and b. The servers in $\mathcal{S}^\ell$ and $\mathcal{S}^{\ell+1}$ then participate in a linear protocol π_{trans} where the inputs of the servers in $\mathcal{S}^\ell$ are the shares computed above.*

3. ***multiplication by constant gate.*** *For every multiplication by constant gate G^c in C with inputs G^a and constant b, Π evaluates G^c by having the servers in $\mathcal{S}^\ell$ multiply its shares corresponding to the outputs of G^a with b. The servers in $\mathcal{S}^\ell$ and $\mathcal{S}^{\ell+1}$ then participate in a linear protocol π_{trans} where the inputs of the servers in $\mathcal{S}^\ell$ are the shares computed above.*

4. ***multiplication gate.*** *For every multiplication gate G^c in C with inputs G^a and G^b, the servers in $\mathcal{S}^\ell$ and $\mathcal{S}^{\ell+1}$ participate in a linear protocol π_{mult} where the inputs of the servers in $\mathcal{S}^\ell$ are the shares of G^a and G^b.*

5. ***relay gate.*** *For every relay gate* G^c *in* C *with input* G^a*,* Π *evaluate* G^c *by considering the corresponding share of* G^a*. The servers in* $\mathcal{S}^\ell$ *and* $\mathcal{S}^{\ell+1}$ *then participate in a linear protocol* π_{trans} *where the inputs of the servers in* $\mathcal{S}^\ell$ *are the shares computed above.*

Output Stage. *The output recovery phase is done as follows. For each output gate of* C*, the first* $t+1$ *clients send their corresponding shares to all other parties, and all the parties in turn recover each output of* C *using* reconstruct.

Note in the last epoch of the execution stage $\mathcal{S}^{d+1} = \mathcal{C}$. Therefore, at the end of the execution stage every client in has a share of the output wires. It's obvious from the description, but is used in the malicious compiler.

Remark. As defined above, each epoch in the execution stage comprises of multiple parallel executions of various linear protocols and each linear protocol consists of a computation phase, a hand-off phase and an output phase. The computation phases of each of the linear protocols in a given epoch are part of the computation phase of that epoch. The hand-off phases of each of these linear protocols together constitute the hand-off phase of that epoch. And the output phases of the linear protocols of a given epoch can be combined with computation phase of the next epoch. A linear-based Fluid MPC protocol is said to have maximal fluidity if it only comprises of maximally fluid linear protocols.

4.5.2 Weak Privacy and Security up to Additive Attacks

We now formalize the notion of *weak privacy* against malicious R-adaptive adversaries. As discussed earlier, a protocol is said to be weakly private if its truncated view in the real execution can be simulated by a simulator in the ideal world. When considering weak privacy in the Fluid MPC setting against a malicious R-adaptive adversary, we must also keep track of the list of all the corrupted servers in each epoch (similar to the security definition in Section 4.2.3). Therefore, we consider the following modified variant of the above definition.

Definition 16 (Weak Privacy). *Let π be a Fluid MPC protocol (with E epochs) for computing a functionality f, and let $\mathcal{A}$ be a malicious R-adaptive adversary, who corrupts a subset $(\mathcal{A} \cap \mathcal{C}) \subset [m]$ of the clients and a subset $\mathcal{A}^{\ell} \subset [n_{\ell}]$ of the servers in each epoch ℓ servers. Denote by $\mathsf{view}_{\mathcal{A}}^{\pi,\mathsf{trunc}}(\overrightarrow{x})$ the view of $\mathcal{A}$ excluding the last communication round*[6] *during a real execution of π on inputs $\overrightarrow{x}$. We say that π is weakly-private against $\mathcal{A}$ if there exists a simulator* Sim *such that,*

$$\left(\mathsf{view}_{\mathcal{A}}^{\pi,\mathsf{trunc}}(\overrightarrow{x}), \{\mathcal{A}^{\ell}\}_{\ell \in [E]}\right) \equiv \left(\mathsf{Sim}(\overrightarrow{x}_{(\mathcal{A} \cap \mathcal{C})}), \{\mathsf{corrupt}^{\ell}\}_{\ell \in [E]}\right)$$

where Sim *gets the following "limited" communication access to the trusted party: The trusted party initializes $\ell = 1$. Until* Sim *indicates the end of the execution stage, the following steps are executed:*

1. *Trusted party sends ℓ to* Sim *and initializes an* append-only *list* $\mathsf{Corrupt}^{\ell}$ *to be $\emptyset$.*

2. Sim *then sends pairs of the form (j, i) where j denotes epoch number and i denotes the* index of the corrupted server *in epoch $j \leq \ell$. Upon receiving this, the trusted party appends i to the list* $\mathsf{Corrupt}^{j}$*. This step can be repeated multiple times.*

3. Sim *sends continue to the trusted party, and the trusted party increments ℓ by 1.*

Sim *can also send an abort message to the trusted party in which case the trusted party outputs the lists that have been updated until the point the abort message was received. Else,* Sim *sends next phase to the trusted party to indicate the end of the execution stage, and hence the end of the corruption phase of servers. In this case, the ideal functionality outputs the final version of $\{\mathsf{corrupt}^{\ell}\}_{\ell \in [E]}$. Notice that* Sim *can only update the trusted functionality f with the list of corrupt servers and cannot make any other queries to the trusted functionality regarding the output of f.*

We now proceed to formalize the notion of additive attacks.

Additive Attack. Let C be a circuit. An *additive attack* A on C assigns a field element to every intermediate wire as well as to the outputs of C. We use $A_{a,b}$ to denote the attack restricted to wire

[6] We emphasize that we are talking about the the last round and not the last epoch here. In any Fluid MPC protocol, this will generally correspond to the last round of the output stage. In other words, this truncated view includes the view of the adversary in the input stage, execution stage (all E epochs) and all but the last round of the output stage.

(a, b), where a and b denote gates. Similarly we use A_{out} to denote the restriction of A to the outputs of C. An additive attack changes the computation performed by circuit C in the following manner. For every wire (a, b) in C, the value $A_{a,b}$ is added to the output of a before it enters the input of b. Similarly the value A_{out} is added to the outputs of C.

Definition 17 (Additively Corruptible Version of a Circuit). *Let $C : (\mathbb{F}^m)^m \to \mathbb{F}^m$ be an m-party circuit containing ω wires. We define the additively corruptible version of C to be the n-party functionality $\widetilde{f}_C : (\mathbb{F}^m)^m \times \mathbb{F}^\omega \to \mathbb{F}^m$ that apart from the inputs $\vec{x}$, takes additional input A from the adversary specifying an* additive attack *for every wire of C, and outputs the result of the additively corrupted C as specified by the additive attack A.*

With the appropriate definitions in place, we can now restate the appropriately modified theorem from [GIP+14] in the context of our setting.

Theorem 2. *Let Π be a Fluid MPC protocol computing a (possibly randomized) n-client circuit $C : (\mathbb{F}^m)^n \to \mathbb{F}^m$ using N servers that is a linear-based Fluid MPC with respect to a t-out-of-n secret sharing scheme, and is weakly-private against malicious R-adaptive adversaries controlling at most $t_\ell < n_\ell/2$ servers in committee $\mathcal{S}_\ell$ (for each $\ell \in [d]$) and $t < m/2$ clients, where d is the depth of the circuit C and n_ℓ are the number of servers in epoch ℓ. Then, Π is a $1/2$-Fluid MPC with R-Adaptive Security with d epochs for computing the additively corruptible version $\widetilde{f}_C$ of C.*

The proof extends identically as in [GIP+14], but for the sake of completeness, we now provide a description of the simulator.

In order to prove Theorem 2, we need to construct a simulator that can "extract" the additive errors induced by the adversary on each intermediate wire. While the view of the adversary until the last round can be simulated using the simulator for weak privacy, the last round messages and the output of the honest parties crucially depend on these additive errors. At a high level, in [GIP+14], the simulator for additive security Sim proceeds as follows: First, Sim invokes the adversary $\mathcal{A}$ on the truncated view simulated by the simulator for weak privacy $\widetilde{\mathsf{Sim}}$. Recall that the truncated view produced by $\widetilde{\mathsf{Sim}}$ consists of the simulated honest party messages, which are relayed from Sim to $\mathcal{A}$

at each step of the protocol, and the corresponding responses from $\mathcal{A}$ are recorded. Next, at each step Sim determines the messages that $\mathcal{A}$ should have sent were it behaving in an honest manner. Using the observation from Claim 1, Sim uses both (a) messages sent by $\mathcal{A}$; and (b) messages that $\mathcal{A}$ should have sent were it behaving honestly; to determine the additive errors injected by $\mathcal{A}$ on each wire. Finally, Sim invokes the ideal functionality, on (a) the inputs extracted from $\mathcal{A}$; and (b) the additive errors for each wire in the circuit. Upon receiving the corresponding output from the ideal functionality, Sim then simulates the messages of the last round appropriately.

Given a simulator for weak privacy against a malicious R-adaptive adversary, the simulator for security up to additive attacks in the Fluid MPC setting works exactly like the simulator described in [GIP+14] for the static corruption setting. This is because, all the messages sent to the adversary until the last round are simulated using the simulator for weak privacy, and extraction of additive errors during these rounds does not affect the view of the adversary. Recall that in the Fluid MPC setting, by corrupting a server in a given epoch, a malicious R-adaptive adversary cannot change the messages that it had sent in any of the prior epochs. Therefore, the additive errors determined by the simulator based on adversary's messages in any given epoch do not change if the adversary decides to corrupt a server at a later stage and can be extracted in a similar way. The last round messages in the Fluid MPC setting, correspond to the messages exchanged by the clients in the output stage. Since the clients are statically corrupted, the same approach can be used to simulate these messages in the Fluid MPC setting as well. Moreover the list of corrupted servers that the simulator for security up to additive attacks is required to send to the trusted functionality can also be determined using the simulator for weak privacy (see Definition 16).

Since we use slightly different notations, for the sake of completeness, we formally describe the simulator. However, we omit the argument for indistinguishability. This is because indistinguishability of the list of corrupt servers and of the adversary's view up to the last round follows from weak privacy. The indistinguishability of the output of the honest clients and the view of the adversary in the last round (i.e., the output computation) depends on whether or not the additive errors were correctly computed by the simulator. Since a malicious R-adaptive adversary cannot change these

errors by corrupting servers at a later stage, this is no different than the static corruption setting. For simplicity, we assume that the number of clients and the number of servers in each epoch are n.

Additionally, we also assume that the adversary corrupts exactly t servers in each epoch. While in reality an adversary could corrupt fewer than t servers in an epoch. This distinction between these two kinds of adversaries has already been studied in the regular MPC setting in [GIP+14]. At a high level they prove this by taking an adversary that corrupts fewer than t parties and suitably augmenting it to construct an adversary that corrupts exactly t parties. Using the intuition that the adversary cannot affect messages previously sent by the honest parties, this idea can also be extended to our Fluid MPC setting. We refer the reader to [GIP+14] for more details.

Simulator Let Π be a linear-based fluid MPC protocol for computing a (possibly) randomized m-client circuit $C : (\mathbb{F}^m)^m \to \mathbb{F}^m$ using $\overrightarrow{n}$ servers that is weakly private against malicious adversaries controlling at most t servers in each epoch, and linear based with respect to a t-out-of-n threshold secret sharing scheme. In addition let $\mathcal{A}$ be an adversary controlling a subset $(\mathcal{A} \cap \mathcal{C})$ of clients and a subset $\mathcal{A}$ of servers. We use $(\mathcal{H} \cap \mathcal{C})$ to denote the set of honest clients. Since an R-adaptive adversary can adaptively corrupt the servers at any point, in the context of this simulator, we use $\mathcal{A}^\ell$ to denote the set of corrupt servers in epoch ℓ during epoch ℓ. This does not include the servers in epoch ℓ that the adversary might choose to corrupt in a later epoch. Similarly, we use $(\mathcal{H} \cap \mathcal{S}^\ell)$ to denote the set of honest servers in epoch ℓ during epoch ℓ. The simulator Sim on input $\overrightarrow{x}_{(\mathcal{A} \cap \mathcal{C})}$, of the corrupted clients, initializes an additive attack A and does the following:

1. **Truncated view generation phase.** Let $\mathsf{Sim}_{\mathsf{trunc\text{-}view}}$ be a simulator guaranteed by the weak-privacy property of Π against malicious R-adaptive adversary. Invoke $\mathsf{Sim}_{\mathsf{trunc\text{-}view}}$ on the inputs $\overrightarrow{x}_{(\mathcal{A} \cap \mathcal{C})}$ and obtain a simulated truncated view $u'_{\mathcal{A}}$. At each step when $\mathsf{Sim}_{\mathsf{trunc\text{-}view}}$ generates an updated list of corrupted servers, Sim forwards it to its trusted functionality.

2. **Input Stage (Random Input Gates).** Let $\mathsf{out}_{(\mathcal{H} \cap \mathcal{S}^1), \pi_{\mathsf{rand}}}$ be the output function of $(\mathcal{H} \cap \mathcal{S}^1)$ in π_{rand} as defined in Definition 14. The simulation proceeds as follows:

   (a) Simulate the honest behavior of the clients in $(\mathcal{A} \cap \mathcal{C})$ given their truncated view $u'_{\mathcal{A}}$ and

obtain the messages $\mathsf{m}'^{\pi_{\mathsf{rand}}}_{(\mathcal{A}\cap\mathcal{C})\to(\mathcal{H}\cap\mathcal{S}^1)}$ that should have been sent by the clients in $(\mathcal{A}\cap\mathcal{C})$ to $(\mathcal{H}\cap\mathcal{S}^1)$ during the execution of π_{rand}. In addition, for every server $P_i \in \mathcal{A}^1$, for every randomness gate G^c obtain the share $\overline{\mathsf{G}}'^c_i$ that is part of the output of P_i at the end of the honest execution of π_{rand}.

(b) Invoke $\mathcal{A}$ on the truncated view $u'_{\mathcal{A}}$ and obtain the messages $\widetilde{\mathsf{m}}'^{\pi_{\mathsf{rand}}}_{(\mathcal{A}\cap\mathcal{C})\to(\mathcal{H}\cap\mathcal{S}^1)}$ sent by the adversary to the servers in $(\mathcal{H}\cap\mathcal{S}^1)$ during the execution of π_{rand}.

(c) Compute $\gamma^{\pi_{\mathsf{rand}}}_{(\mathcal{H}\cap\mathcal{S}^1)} \leftarrow \mathsf{out}_{(\mathcal{H}\cap\mathcal{S}^1),\pi_{\mathsf{rand}}}(0, \widetilde{\mathsf{m}}'^{\pi_{\mathsf{rand}}}_{(\mathcal{A}\cap\mathcal{C})\to(\mathcal{H}\cap\mathcal{S}^1)} - \mathsf{m}'^{\pi_{\mathsf{rand}}}_{(\mathcal{A}\cap\mathcal{C})\to(\mathcal{H}\cap\mathcal{S}^1)})$.

(d) For every randomness gate G^c, let $\gamma^c_{(\mathcal{H}\cap\mathcal{S}^1)} \in \mathbb{F}^{t+1}$ be the restriction of $\gamma^{\pi_{\mathsf{rand}}}_{(\mathcal{H}\cap\mathcal{S}^1)}$ to the values corresponding to G^c.

    i. The simulator now determines entries for the additive attack A on the circuit C. Notice that $\gamma^c_{(\mathcal{H}\cap\mathcal{S}^1)}$ is a vector of $t+1$ shares of the threshold secret sharing scheme, and thus forms a valid sharing of some value.

    Compute $\alpha^c \coloneqq \mathsf{reconstruct}(\gamma^c_{(\mathcal{H}\cap\mathcal{S}^1)}, (\mathcal{H}\cap\mathcal{S}^1))$, and for every gate G^d connected to G^c set $A_{c,d} \coloneqq \alpha^c$. Additionally, compute the shares $\gamma^c_{\mathcal{A}^1}$ of the adversarial servers consistent with $\gamma^c_{(\mathcal{H}\cap\mathcal{S}^1)}$.

    ii. The simulator for each $P^1_i \in \mathcal{A}^1$ computes the share $\mathsf{G}'^c_i \coloneqq \overline{\mathsf{G}}'^c_i + \gamma^c_i$.

3. **Input Stage (Input Gates).**

(a) for each input gate G^c that is part of the inputs of some honest client P_i:

    i. for every corrupted server P^1_j, retrieve from $u'_{\mathcal{A}}$ the value G'^c_j representing P^1_j's share of P_i's input for G^c and send it to $\mathcal{A}$.

    ii. for any gate G^d connected to the output of G^c, set $A_{c,d} \coloneqq 0$.

(b) For each input gate G^c that is part of the inputs of some adversarial client P_i:

    i. for each honest server P^1_j, receive a message $\widetilde{\mathsf{G}}'^c_j$ from $\mathcal{A}$ corresponding to the P^1_j's share of $\mathcal{A}$'s input for G^c.

    ii. notice that the honest shares is a vector of $t+1$ shares of the threshold secret sharing scheme, and thus forms a valid sharing of some value.

Compute $\widetilde{x}^c := \mathsf{reconstruct}(\left\{\widetilde{\mathsf{G}}_j'^c\right\}_{P_j^1 \in (\mathcal{H} \cap \mathcal{S}^1)}, (\mathcal{H} \cap \mathcal{S}^1))$. for any gate G^d connected to the output of G^c, set $A_{c,d} := \widetilde{x}^- x_c$ where x_c is the input of P_i to G^c.

iii. For each corrupted server compute the shares $\mathsf{G}^c_{\mathcal{A}^1}$ of the adversarial servers consistent with the shares obtain above.

4. **Execution Stage.** For each layer $\ell \in [d]$, the simulator simulates all the gates in the layer ℓ as follows

Addition gate. For each corrupted server, do the following:

(a) Simulate the honest behavior of the servers in $\mathcal{A}^\ell$ given their truncated view $u'_\mathcal{A}$, on main inputs $(\mathsf{G}_i'^a + \mathsf{G}_i'^b)_{P_i^\ell \in \mathcal{A}^\ell}$ and obtain the messages $\mathsf{m}'^{\pi_{\mathsf{trans}}}_{\mathcal{A}^\ell \to (\mathcal{H} \cap \mathcal{S}^\ell)}$ that should have been sent by the servers in $\mathcal{A}^\ell$ to $(\mathcal{H} \cap \mathcal{S}^\ell)$ during the execution of π_{trans}.

In addition, for every server $P_i^{\ell+1} \in \mathcal{A}^{\ell+1}$, obtain the share $\overline{\mathsf{G}}_i'^c$ that is part of the output of $P_i^{\ell+1}$ at the end of the execution of π_{trans}.

(b) Invoke $\mathcal{A}$ on the truncated view $u'_\mathcal{A}$ and obtain the messages $\widetilde{\mathsf{m}}'^{\pi_{\mathsf{trans}}}_{\mathcal{A}^\ell \to (\mathcal{H} \cap \mathcal{S}^\ell)}$ sent by the adversary to the servers in $(\mathcal{H} \cap \mathcal{S}^\ell)$ during the execution of π_{rand}.

(c) Compute $\delta^{\pi_{\mathsf{trans}}}_{(\mathcal{H} \cap \mathcal{S}^\ell)} \leftarrow \mathsf{out}_{(\mathcal{H} \cap \mathcal{S}^\ell), \pi_{\mathsf{trans}}}(0, \widetilde{\mathsf{m}}'^{\pi_{\mathsf{trans}}}_{\mathcal{A}^\ell \to (\mathcal{H} \cap \mathcal{S}^\ell)} - \mathsf{m}'^{\pi_{\mathsf{trans}}}_{\mathcal{A}^\ell \to (\mathcal{H} \cap \mathcal{S}^\ell)})$.

(d) The simulator now determines entries for the additive attack A on the circuit C. Notice that $\delta^c_{(\mathcal{H} \cap \mathcal{S}^\ell)}$ is a vector of $t+1$ shares of the threshold secret sharing scheme, and thus forms a valid sharing of some value.

Compute $\alpha^c := \mathsf{reconstruct}(\delta^c_{(\mathcal{H} \cap \mathcal{S}^\ell)}, (\mathcal{H} \cap \mathcal{S}^\ell))$, and for every gate G^d connected to G^c set $A_{c,d} := \alpha^c$. Additionally, compute the shares $\delta^c_{\mathcal{A}^\ell}$ of the adversarial servers consistent with $\delta^c_{(\mathcal{H} \cap \mathcal{S}^\ell)}$.

(e) The simulator for each $P_i^{\ell+1} \in \mathcal{A}^{\ell+1}$ computes the share $\mathsf{G}_i'^c := \overline{\mathsf{G}}_i'^c + \delta_i^c$.

Addition-by-a-constant and multiplication-by-a-constant gates. The simulation proceeds identically as above with the only change being that simulation of the honest behavior of the

adversarial servers are done with inputs $(\mathsf{G}'^a_i + b)_{P^\ell_i \in \mathcal{A}^\ell}$ (respectively $(\mathsf{G}'^a_i \cdot b)_{P^\ell_i \in \mathcal{A}^\ell}$) for the addition-by-a-constant (respectively multiplication-by-a-constant) gate.

Relay gate. As above, the simulation is identical to the addition gate with the only change being that simulation of the honest behavior of the adversarial servers are done with inputs G^a for the relay gate.

Multiplication gate. As above, the simulation is identical to the addition gate with the following two changes:

(a) the simulation is done for the protocol π_{mult} instead of π_{trans}; and

(b) the inputs to π_{mult} are $(\mathsf{G}'^a_i, \mathsf{G}'^b_i)_{P^\ell_i \in \mathcal{A}^\ell}$.

5. **Output stage.** At the end of the circuit evaluation phase, for each output gate G^z each corrupted client $P_i \in (\mathcal{A} \cap \mathcal{C})$ holds a share $\widetilde{\mathsf{G}}^z_i$ of the supposed output.

   (a) The simulator sets to 0 all coordinates of A that were not previously set.

   (b) The simulator invokes the trusted party computing $\widetilde{f}_C$ with the inputs of the corrupted parties and with the aforementioned wire corruptions A. The trusted party responds to the simulator with the output y.

   (c) For each output gate G^z of C that is connected to an output of some gate g^a the simulator chooses shares of y_z that are compatible with $(\mathsf{G}^a_i)_{P^\in_i (\mathcal{A} \cap \mathcal{C})}$, adds them to $u'_\mathcal{A}$ and sends them to $\mathcal{A}$.

   (d) The simulator outputs $u'_\mathcal{A}$.

The proof of indistinguishability follows identically as in [GIP+14], and we refer the reader to their paper for further details.

4.6 Malicious Security Compiler for Fluid MPC

In this section, we describe a generic compiler that can compile any linear-based Fluid MPC protocol that is secure up to additive attacks against a malicious R-adaptive adversary into one that achieves security with abort against R-adaptive adversaries (Definition 10) in the fluid MPC setting. Our compiler achieves two main properties: (1) it preserves the fluidity of the underlying protocol and (2) only incurs a constant multiplicative overhead in the communication complexity of the underlying protocol. We discuss these properties in detail in the upcoming subsections.

As discussed in Section 4.1, in order to go from security up to additive attacks to security with abort against malicious adversaries, we require the parties to compute a MAC of each individual wire value and incrementally compute two random linear combinations: (1) one using the actual values induced on the intermediate wires of the circuit during evaluation and (2) the other one using the MAC values corresponding to these wire values. Finally, correctness of the computation is verified by performing a check on the two linear combinations. For designing a generic compiler that implements this idea, we proceed in two main steps.

1. In the first step (Section 4.6.1), given a layered arithmetic circuit C, we augment it to obtain a *robust circuit* $\tilde{C}$, that additionally computes these MAC values and the two linear combinations.

2. Then, in the second step (Section 4.6.2), we run the underlying protocol (say Π) that is secure up to additive attacks on this robust circuit $\tilde{C}$. Before executing the output stage of Π, the clients first check if the computation was done honestly by comparing the two linear combinations. They proceed to the output stage of Π only if this check succeeds.

From the previous section, we know that any weakly private linear-based Fluid MPC is secure against a malicious R-adaptive adversary up to additive attacks. Hence, for the remainder of this section, we refer to the underlying linear-based Fluid MPC as being weakly private or being secure against a malicious R-adaptive adversary, interchangeably. For simplicity, throughout this section, we assume that the number of clients and number of servers in each committee are n. While in most places it is easy to see how the protocol can be extended to support committees of different sizes, we

add additional remarks wherever necessary. We also assume (w.l.o.g.) that all parties get the same output.

4.6.1 Robust Circuit

In this section, we describe the first step towards building our malicious security compiler, i.e., transforming a layered circuit C into a robust circuit $\tilde{C}$. We transform C in such a way, that the resulting circuit $\tilde{C}$ computes the two linear combinations (mentioned above) incrementally. Recall that this incremental computation is necessary in order to prevent the size of the circuit from blowing up. As a result, our transformation only incurs a constant (multiplicative) overhead in the size of the original circuit C. Another property of our transformation is that the resulting protocol is also a layered circuit.[7]

We start by formally defining a robust circuit.

Definition 18 (Robust Circuit). *Given a layered arithmetic circuit C for functionality f of depth d and maximum width w, the robust circuit $\tilde{C}$ corresponding to C, that realizes a functionality $\tilde{f}$ that computes the following:*

1. *Original Output: Compute $\overrightarrow{z} = C(\overrightarrow{x})$ on the given set of inputs $\overrightarrow{x}$.*

2. *Random Values: Sample random values $r \in \mathbb{F}$, $\beta \in \mathbb{F}$ and $\alpha_1, \ldots, \alpha_w \in \mathbb{F}^w$.*

3. *Linear Combinations: Computes the following linear combinations*

$$u = \sum_{l=0}^{d} \left(\sum_{k=1}^{w} \alpha_k^l z_k^l \right) \text{ and } v = \sum_{l=0}^{d} \left(\sum_{k=1}^{w} \alpha_i^l (r z_k^l) \right)$$

   where z_k^ℓ corresponds to the output of gate G_k^ℓ (k^{th} gate on level ℓ), $\alpha_k^0 = \alpha_k$ and for $\ell > 0$

$$\alpha_k^\ell = \alpha_k^{\ell-1} \beta = \alpha_k (\beta)^\ell$$

4. *Final Output: Output $\overrightarrow{z}, r, u, v$.*

[7] This property is necessary for the second step in our compiler and reason behind it will become clear in Section 4.6.2.

We now show how any layered circuit can be transformed into a robust circuit with constant overhead in size.

Lemma 10. *Any layered arithmetic circuit C for functionality f with depth d and maximum width w, can be transformed into a randomized layered robust circuit $\tilde{C}$ for functionality $\tilde{f}$ (as defined in 18) of depth $d+1$ and maximum width $4w+4$.*

Proof. The transformation proceeds as follows:

1. Add $w+2$ random input gates for $r, \alpha_1, \ldots, \alpha_w, \beta \in \mathbb{F}$ on level $\ell = 0$.

2. Add n multiplication gates on level 1 to multiply each of the input values $\{x_i\}_{i \in [n]}$ with the random input r.

3. All the gates in on level $\ell > 0$ in the original circuit C, are now on level $\ell + 1$. Add relay gates on level $\ell = 1$ to connect the input gates with the gates on level $\ell = 2$ (note that these gates were originally on level $\ell = 1$).

4. Now for each layer $\ell \in \{2, \ldots, d+1\}$, do the following:

   – For each gate G_k^ℓ (for $k \in [w]$), do the following:

   – **If G_k^ℓ is an addition gate:** Let G_k^ℓ take as input a set of values $\{z_i^{\ell-1}\}_{i \in Q}$ from the previous layer, add another addition gate on layer ℓ with a similar in-degree that takes as input values $\{r z_i^{\ell-1}\}_{i \in Q}$.

   – **If G_k^ℓ is a multiplication gate:** Let G_k^ℓ take as input values $z_i^{\ell-1}, z_j^{\ell-1}$ from the previous layer, add another multiplication gate on layer ℓ that takes as input values $r z_i^{\ell-1}$ and $z_j^{\ell-1}$.

   – **If G_k^ℓ is a multiplication-by-constant gate:** Let G_k^ℓ take as input value $z_i^{\ell-1}$ from the previous layer, add another multiplication-by-constant gate on layer ℓ that takes as input value $r z_i^{\ell-1}$.

   – **If G_k^ℓ is an addition-by-constant gate:** Let G_k^ℓ take as input value $z_i^{\ell-1}$ from the previous layer and has a value c hard-wired in it, add a multiplication-by-constant gate on level $\ell - 1$ that has the value c hardwired in it and takes as input r. Add another addition gate on layer

ℓ that takes as input value $rz_i^{\ell-1}$ and the output of the new multiplication-by-constant gate on level $\ell - 1$.

  – **If G_k^ℓ is a relay gate:** Let G_k^ℓ take as input $z_i^{\ell-1}$ from the previous layer, add another relay gate on layer ℓ with a similar in-degree that takes as input values $rz_i^{\ell-1}$.

– Add $3w$ multiplication gates where the first w gates are used for multiplying $\alpha_k^{\ell-1}$ with β to output α_k^ℓ, The next set of w gates are used for multiplying $\alpha_k^{\ell-1}$ with $z_k^{\ell-1}$ and the last set of w gates are used for multiplying $\alpha_k^{\ell-1}$ with $rz_k^{\ell-1}$.

– If $\ell > 2$, add 2 addition gates to add $u^{\ell-2}, \{\alpha_k^{\ell-2} z_k^{\ell-2}\}_{k\in[w]}$ to get $u^{\ell-1}$, $v^{\ell-2}, \{\alpha_k^{\ell-2} r z_k^{\ell-2}\}_{k\in[w]}$ to get $v^{\ell-1}$ respectively (assuming $u^0 = 0$ and $v^0 = 0$).

– Add 2 relay gates to relay r, β to the next level respectively.

5. At the end the circuit outputs the actual output z of C along with r, $u = u^d$ and $v = v^d$.

□

4.6.2 Maliciously Secure Fluid MPC

In this section, we describe the final step towards building our compiler. Our malicious security compiler, works by running the weakly private linear-based Fluid MPC protocol (say Π) on a robust circuit $\tilde{C}$ (as defined earlier). In the output stage, the clients first check if the computation was done honestly by comparing the linear combinations (computed in the robust circuit). If this check succeeds, the clients reveal the shares of the "actual" outputs and reconstruct the output. Incorporating this additional check to verify correctness of output, bootstraps the security of the underlying protocol to security with abort against malicious R-adaptive adversaries (as defined in definition 10).

It is easy to see that since the execution stage of the weakly private protocol is executed as is (albeit on a different circuit), the resulting protocol achieves the same fluidity as the underlying protocol. Moreover, since the size of the robust circuit on which this underlying protocol is executed is only a constant times bigger than the original layered circuit, our compiler only incurs a constant multiplicative overhead in the communication complexity of the servers.

4.6.2.1 Checking Equality to Zero

We first discuss a functionality described in Chida et.al [CGH+18], that enables a set of parties to check whether the shares held by the parties correspond to a valid sharing of the value 0, without revealing any further information on the shared value. Looking ahead, this functionality will be used in our compiled protocol for the verification check at the end. For the sake of completeness we describe this functionality in figure 4.4. We refer the reader to [CGH+18] for the description of the protocol that securely realizes this functionality.

The functionality $f_{\mathsf{checkZero}}(\mathcal{C} := \{P_1, \ldots, P_n\})$

The n-party functionality $f_{\mathsf{checkZero}}$, running with clients $\{P_1, \ldots, P_n\}$ and the ideal adversary Sim receives $[v]_{\mathcal{H}}$ from the honest clients and uses them to compute v.

- If $v = 0$, then $f_{\mathsf{checkZero}}$ sends 0 to the ideal adversary Sim. If Sim responds with reject (resp., accept), then $f_{\mathsf{checkZero}}$ sends reject (resp., accept) to the honest parties.
- If $v \neq 0$, then $f_{\mathsf{checkZero}}$ procees as follows:
  - With probability $\frac{1}{|\mathbb{F}|}$ it sends accept to the honest clients and ideal adversary Sim.
  - With probability $1 - \frac{1}{|\mathbb{F}|}$ it sends reject to the honest clients and ideal adversary Sim.

Figure 4.4: Functionality for checking equality to zero

Lemma 11. *[CGH+18] There exists a protocol that securely realizes $f_{\mathsf{checkZero}}$ with abort in the presence of static malicious adversaries who control $t < n/2$ parties.*

Looking ahead, this sub-protocol will be run by the clients in the output stage. We note that it suffices for the protocol realizing $f_{\mathsf{checkZero}}$ to be secure against a *static* malicious adversary because an R-adaptive adversary only statically corrupts the clients.

4.6.2.2 Compiled Protocol

Finally, we describe a Fluid MPC protocol that achieves security with abort against an R-adaptive adversary that can corrupt $t < n/2$ clients and $t < n/2$ servers in each committee in the $f_{\mathsf{checkZero}}$-hybrid model.

Auxiliary Inputs: A finite field $\mathbb{F}$ and a layered robust arithmetic circuit $\tilde{C}$ (corresponding to C) of depth d and width w over $\mathbb{F}$ that computes the function $\tilde{f}$ on inputs of length n.

Parties: The protocol is executed by the following sets of parties: (1) *Clients:* $\mathcal{C} := \{P_1, \ldots, P_n\}$ and (2) *Servers:* For each $\ell \in [d]$, $\mathcal{S}^\ell := \{P_1^\ell, \ldots, P_n^\ell\}$, where d is the depth of the circuit $\tilde{C}$.

Inputs: For each $j \in [n]$, client P_j holds input $x_i \in \mathbb{F}$. All other other parties have no input.

Protocol: Let Π be a weakly private linear-based Fluid MPC protocol. The clients and servers execute the *input and execution stage* of protocol Π for circuit $\tilde{C}$. Let $[z], [r], [u], [v]$ be the shares obtained by the clients at the end of the execution stage. The output stage is modified as follows:

- The clients locally compute: $[T] = [v] - [r] \cdot [u]$
- They invoke $f_{\mathsf{checkZero}}$ on $[T]$. If $f_{\mathsf{checkZero}}$ outputs reject, the clients output $\bot$. Else, if it outputs accept, the clients run the *output stage* reveal their shares of z.

Output: All clients then locally run open$([z])$ to learn the output.

This completes the description of our compiled maliciously secure protocol. We now proceed to analyze its concrete efficiency.

Concrete Efficiency. Let $W_{\mathsf{exec}}(n_{\ell-1}, w, n_\ell)$ be the total communication/computation complexity of epoch ℓ in the weakly private linear-based Fluid MPC protocol, where $n_{\ell-1}$ (and n_ℓ, resp.) is the size of the committee in epoch $\ell - 1$ (and ℓ, resp.) and w is the maximum width of the layered circuit C representing the functionality f. In the above transformation, the layered circuit C of depth d, and width w transformed into a robust layered circuit of depth $d+1$ and width $4w+4$. Running the weakly private linear-based Fluid MPC protocol on this robust circuit, yields the total communication and computation complexity of $W_{\mathsf{exec}}(n_{\ell-1}, (4w+4), n_\ell)$ in epoch ℓ.

Theorem 3. *Let $C : (\mathbb{F}^m)^n \to \mathbb{F}^m$ be a (possibly randomized) m-client circuit. Let $\tilde{C}$ be the robust circuit corresponding to C (see Definition 12). Let Π be a Fluid MPC protocol computing $\tilde{C}$ using N*

servers that is a linear-based Fluid MPC with respect to a t-out-of-n secret sharing scheme, and is weakly-private against malicious R-adaptive adversaries controlling at most $t_\ell < n_\ell/2$ servers in committee $\mathcal{S}_\ell$ (for each $\ell \in [d+1]$) and $t < m/2$ clients, where d is the depth of the circuit C and n_ℓ is the number of servers in epoch ℓ. Then, the above protocol is a $1/2$-Fluid MPC with R-Adaptive Security with $d+1$ epochs for computing C. Moreover, this protocol preserves the fluidity of Π and only adds a constant multiplicative overhead to the communication complexity of Π.

Proof. From Theorem 2, we know that a weakly private linear-based Fluid MPC realizes functionality $\tilde{f}_C$ against malicious R-adaptive adversaries. In other words, it achieves security against such malicious adversaries up to additive attacks, meaning that the adversary can add an arbitrary error value to each wire in the circuit. Since our robust circuit $\tilde{C}$ computes on different types of values, we use different variables to denote the additive errors that the adversary can inject on each of these computations. For simplicity, we assume $\ell \in [0, d]$, where $\ell = 0$ consists of input and random input gates.

- Let ϵ_β^ℓ be the additive error value added by the adversary on the output of the relay gate on level ℓ that is used to transfer β.
- Let $\epsilon_{\alpha,k}^\ell$ be the additive error value added by the adversary on the output of the multiplication gate on level ℓ that is used to multiply $\alpha_k^{\ell-1}$ with β.
- Let ϵ_r^ℓ be the additive value added by the adversary on the output of the relay gate on level ℓ that is used to transfer r. We use ϵ_r to denote $\sum_{\ell=\{0,\ldots,d\}} \epsilon_r^\ell$.
- Let $\epsilon_{z,k}^\ell$ be the additive error value added by the adversary on the output of the kth gate on level ℓ in the original circuit C when evaluated on actual inputs $\overrightarrow{x}$.
- Let $\epsilon_{rz,k}^\ell$ be the additive error value added by the adversary on the output of the kth gate on level ℓ in the original circuit C when evaluated on randomized inputs $r\overrightarrow{x}$.
- Let ϵ_u denote the cumulative errors added on the multiplication gates used to multiply the output of each gate z_k^ℓ with the respective α_k^ℓ and the errors added on the relay gates used to transfer

partially computed values of u at each level.

– Similarly, let ϵ_v denote the cumulative errors added on the multiplication gates used to multiply the output of each gate (on randomized inputs) rz_k^ℓ with the respective α_k^ℓ and the errors added on the relay gates used to transfer partially computed values of v at each level.

Let $\mathcal{A}$ be the real adversary who controls the set of corrupted clients and servers. The simulator Sim works as follows:

Simulator. We describe the simulator in $f_{\mathsf{checkZero}}$- hybrid model. The simulator uses the simulator of the underlying weakly private linear-based Fluid MPC protocol to simulate messages for the adversary in the input stage and execution stage. During simulation, it stores the inputs of the adversarial clients and the additive errors added by the adversary on each wire, that are extracted by the simulator of the underlying protocol. It also forwards the list of corrupt servers sent by the underlying simulator to its ideal functionality. At the end of the execution stage, it performs the following check:

– If there does not exist any non-zero error of the form ϵ_r^ℓ or $\epsilon_{z,k}^\ell$ or $\epsilon_{rz,k}^\ell$ or ϵ_u or ϵ_v,[8] it sends the extracted inputs of the adversarial clients to the ideal functionality and gets the output z. It simulates $f_{\mathsf{checkZero}}$ sending accept to the adversary. Finally, it runs the last step of the underlying simulator on input z to compute the last set of messages for the adversary. It ignores the shares of r, u, v and only forwards the shares of z to the adversary. Upon receiving shares of z from the adversary on behalf of each honest client $P_i \in (\mathcal{H} \cap \mathcal{C})$, it checks if all the shares of z are consistent. If so, it sends $\mathsf{continue}, i$ to the ideal functionality, to instruct it to send the correct output to the honest client C_i. Else, it sends abort, i, in which case the honest client C_i gets $\perp$.

– Else there exists any non-zero error of the form ϵ_r^ℓ or $\epsilon_{z,k}^\ell$ or $\epsilon_{rz,k}^\ell$ or ϵ_u or ϵ_v. It also sends $\perp$ to its ideal functionality. It simulates $f_{\mathsf{checkZero}}$ sending reject to the adversary. The simulator simulates sending $\perp$ to the adversary on behalf of all the honest parties. The output of all the honest parties

[8] We note that the simulator does not need to account for additive errors of the form ϵ_β^ℓ and $\epsilon_{\alpha,k}^\ell$. This is because additive errors on β and the α values does not affect correctness of the "real" output. This point will become clear later in the indistinguishability argument.

is $\perp$ in this case.

Finally, it outputs whatever $\mathcal{A}$ outputs.

Remark. As is clear from the description of the simulator, we argue selective security with abort against R-adaptive adversaries. The security can be easily bootstrapped to unanimous abort (in a straight-forward manner), if the clients have access to a broadcast channel in the last round or if they implement a broadcast over point-to-point channels.

Indistinguishability Argument. We need to argue indistinguishability of the view of the adversary, the outputs of the honest clients and the list of corrupt servers in the real and ideal worlds. Indistinguishability of the list of corrupt servers follows from the security of the underlying protocol up to additive attacks. Next, we note that the only difference between the view generated by the simulator (and how the output of the honest parties is decided) in the ideal world and that obtained in the real execution is that the simulator sends reject on behalf of $f_{\mathsf{checkZero}}$ if it sees any additive errors of the form e^ℓ or $\epsilon^\ell_{z,k}$ or $\epsilon^\ell_{rz,k}$ or ϵ_u or ϵ_v. If $f_{\mathsf{checkZero}}$ returns accept in the real world, then the view generated by the simulator and the output of the honest clients is trivially distinguishable from that of the real execution. We argue that this happens with at most negligible probability.

Recall that if every party behaves honestly, then

$$u = \sum_{l=0}^{d}\left(\sum_{k=1}^{w}\alpha_k^l z_k^l\right) \text{ and } v = \sum_{l=0}^{d}\left(\sum_{k=1}^{w}\alpha_k^l (r z_k^l)\right)$$

We would like to check if $ru = v$, ie.

$$r\left[\sum_{l=0}^{d}\left(\sum_{k=1}^{w}\alpha_k^l z_k^l\right)\right] = \sum_{l=0}^{d}\left(\sum_{k=1}^{w}\alpha_k^l (r z_k^l)\right)$$

This is trivially true if no additive errors were added by the adversary at any step. Accounting for all the additive errors that the adversary might introduce, we get the following, where $\hat{\alpha_k}^0 = \alpha_k^0 + \epsilon_{\alpha,k}^0$

and for $\ell > 0$, $\hat{\alpha_k}^\ell = \hat{\alpha}_k^{\ell-1}(\beta + \sum_{j=0}^{\ell} \epsilon_\beta^j) + \epsilon_{\alpha,i}^\ell$

$$ru = (r + \epsilon_r)\left[\sum_{\ell=0}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}_k^\ell(z_k^\ell + \epsilon_{z,k}^\ell)\right) + \epsilon_u\right] \qquad v = \sum_{\ell=0}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}_k^\ell(rz_k^\ell + \epsilon_{rz,k}^\ell)\right) + \epsilon_v$$

We now consider the following cases:

– **Case 1:** No additive errors introduced in computation of the original circuit on $\overrightarrow{x}$ and $r\overrightarrow{x}$. This does not preclude errors introduced as a consequence of relay gates for r, i.e., $\forall \ell \in \{0, \ldots, d\}$ and $\forall k \in [w]$, $\epsilon_{z,k}^\ell, \epsilon_{rz,k}^\ell = 0$: We want to calculate the probability that the following equation holds, i.e.,

$$(r + \epsilon_r)\left[\sum_{\ell=0}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}_k^\ell z_k^\ell\right) + \epsilon_u\right] = \sum_{\ell=0}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}_k^\ell r z_k^\ell\right) + \epsilon_v$$

in other words

$$r\epsilon_u = \epsilon_v - \epsilon_r\left[\sum_{\ell=0}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}_k^\ell z_k^\ell\right) + \epsilon_u\right]$$

– **Case a:** If $\epsilon_u \neq 0$

Since r is sampled uniformly, the probability that the following holds is $1/|\mathbb{F}|$.

$$r = \left(\epsilon_v - \epsilon_r\left[\sum_{\ell=0}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}_k^\ell z_k^\ell\right) + \epsilon_u\right]\right) \cdot \epsilon_u^{-1}$$

– **Case b:** Else if $\epsilon_u = 0$, then

$$\epsilon_v = \epsilon_r\left[\sum_{\ell=0}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}_k^\ell z_k^\ell\right)\right] \tag{4.1}$$

We know that $\hat{\alpha_k}^\ell = \hat{\alpha}_k^{\ell-1}(\beta + \sum_{j=0}^{\ell} \epsilon_\beta^j) + \epsilon_{\alpha,k}^\ell$, we expand each $\hat{\alpha_k}^\ell$ and write it out as terms that depend on α_0 and terms that don't

$$\hat{\alpha_k}^\ell = p_k^\ell + \left(\alpha_k^0 \prod_{j=0}^{\ell}(\beta + \sum_{i=0}^{j}\epsilon_\beta^i)\right)$$

where p_k^ℓ only depends on β and the additive errors added but not on α_k^0 and can be expanded

as the following:

$$p_k^\ell = \hat{\alpha}_k^{\ell-1}(\beta + \sum_{j=0}^{\ell} \epsilon_\beta^j) + \epsilon_{\alpha,k}^\ell - \left(\alpha_k^0 \prod_{j=0}^{\ell} (\beta + \sum_{i=0}^{j} \epsilon_\beta^i) \right)$$

Let $q \in [w]$ be the smallest q such that $\exists z_q^\ell \neq 0$ for some $\ell \in \{0, \ldots, d\}$. From equation 4.1, ϵ_v is equal to the following:

$$\begin{aligned}
\epsilon_v &= \epsilon_r \left[\sum_{\ell=0}^{d} \hat{\alpha}_q^\ell + \sum_{\ell=0}^{d} \left(\sum_{k=1,k\neq q}^{w} \hat{\alpha}_k^\ell z_k^\ell \right) \right] \\
&= \epsilon_r \left[\sum_{\ell=0}^{d} p_q^\ell z_q^\ell + \alpha_q^0 \sum_{\ell=0}^{d} \prod_{j=0}^{\ell} (\beta + \sum_{i=0}^{j} \epsilon_\beta^i) z_q^\ell + \sum_{\ell=0}^{d} \left(\sum_{k=1,k\neq q}^{w} \hat{\alpha}_k^\ell z_k^\ell \right) \right]
\end{aligned}$$

Which can be rewritten as

$$\alpha_q^0 \epsilon_r \left(\sum_{\ell=0}^{d} \prod_{j=0}^{\ell} (\beta + \sum_{i=0}^{j} \epsilon_\beta^i) z_q^\ell \right) = \left(\epsilon_v - \epsilon_r \left[\sum_{\ell=0}^{d} p_q^\ell z_q^\ell + \sum_{\ell=0}^{d} \left(\sum_{k=1,k\neq q}^{w} \hat{\alpha}_k^\ell z_k^\ell \right) \right] \right) \quad (4.2)$$

We now consider the following two cases:

1. If $\epsilon_r \left(\sum_{\ell=0}^{d} \prod_{j=0}^{\ell} (\beta + \sum_{i=0}^{j} \epsilon_\beta^i) z_q^\ell \right) = 0$:

   Then either $\epsilon_r = 0$, which from equation 4.1 would imply that $\epsilon_v = 0$. This would mean that the adversary has only injected additive errors on the computations and transfers of α's and β. This does not hamper the correctness of output.

   Else, this is a uni-variate polynomial in β with degree at most d. Such a polynomial has at most d roots. Since β is uniformly distributed, the probability that β is equal to one of these roots is $d/|\mathbb{F}|$.

2. Else if $\epsilon_r \left(\sum_{\ell=0}^{d} \prod_{j=0}^{\ell} (\beta + \sum_{i=0}^{j} \epsilon_\beta^i) z_q^\ell \right) \neq 0$:

   Since α_q^0 is uniformly distributed, the probability that the equality in Equation 4.2 holds is $1/|\mathbb{F}|$.

Hence, overall the probability that that the view generated by the simulator in Case 1 is distin-

guishable from the view in the real execution is at most

$$\frac{1}{|\mathbb{F}|} + \left(1 - \frac{1}{|\mathbb{F}|}\right)\left(\frac{d}{|\mathbb{F}|} + \left(1 - \frac{d}{|\mathbb{F}|}\right)\frac{1}{|\mathbb{F}|}\right) < \frac{d+1}{|\mathbb{F}|}$$

– **Case 2:** Not all $\epsilon^{\ell}_{z,k}$ and $\epsilon^{\ell}_{rz,k}$ are 0: Let the qth gate on level m be the first gate with non-zero errors. We want to calculate the probability that $ru = v$, where:

$$\begin{aligned}
ru = {} & (r+\epsilon_r)\left[\sum_{\ell=0}^{m-1}\left(\sum_{k=1}^{w}\hat{\alpha}^{\ell}_k z^{\ell}_k\right) + \sum_{k=1}^{q-1}\hat{\alpha}^m_k z^m_k\right] + (r+\epsilon_r)\left[\hat{\alpha}^m_q(z^m_q + \epsilon^m_{z,q}) + \sum_{k=q+1}^{w}\hat{\alpha}^m_k(z^m_k + \epsilon^m_{z,k})\right] \\
& + \ (r+\epsilon_r)\left[\sum_{\ell=m+1}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}^{\ell}_k(z^{\ell}_k + \epsilon^{\ell}_{z,k})\right) + \epsilon_u\right] \\
v = {} & \sum_{\ell=0}^{m-1}\left(\sum_{k=1}^{w}\hat{\alpha}^{\ell}_k r z^{\ell}_k\right) + \sum_{k=1}^{q-1}\hat{\alpha}^m_k r z^m_k + \hat{\alpha}^m_q(r z^m_q + \epsilon^m_{rz,q}) + \sum_{k=q+1}^{w}\hat{\alpha}^m_k(r z^m_k + \epsilon^m_{rz,k}) \\
& + \sum_{\ell=m+1}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}^{\ell}_k(r z^{\ell}_k + \epsilon^{\ell}_{rz,k})\right) + \epsilon_v
\end{aligned}$$

Substituting into $ru = v$, and canceling the equal terms (similar to Case 1) we get

$$\begin{aligned}
\hat{\alpha}^m_q\left(\epsilon_r(z^m_q + \epsilon^m_{z,q}) - \epsilon^m_{rz,q} + r\epsilon^m_{z,q}\right) = {} & \sum_{k=q+1}^{w}\hat{\alpha}^m_k\epsilon^m_{rz,k} + \sum_{\ell=m+1}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}^{\ell}_k\epsilon^{\ell}_{rz,k}\right) + \epsilon_v \\
& - \ r\left[\sum_{k=q+1}^{w}\hat{\alpha}^m_k\epsilon^m_{z,k} + \sum_{\ell=m+1}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}^{\ell}_k\epsilon^{\ell}_{z,k}\right) + \epsilon_u\right] \\
& - \ \epsilon_r\left[\sum_{\ell=0}^{m-1}\left(\sum_{k=1}^{w}\hat{\alpha}^{\ell}_k z^{\ell}_k\right) + \sum_{k=1}^{q-1}\hat{\alpha}^m_k z^m_k + \sum_{k=q+1}^{w}\hat{\alpha}^m_k(z^m_k + \epsilon^m_{z,k}) + \sum_{\ell=m+1}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}^{\ell}_k(z^{\ell}_k + \epsilon^{\ell}_{z,k})\right) + \epsilon_u\right]
\end{aligned}$$

This can be further simplified to get

$$\begin{aligned}
\hat{\alpha}^m_q\left(\epsilon_r(z^m_q + \epsilon^m_{z,q}) - \epsilon^m_{rz,q} + r\epsilon^m_{z,q}\right) = {} & \epsilon_v - (r+\epsilon_r)\epsilon_u + \sum_{k=q+1}^{w}\hat{\alpha}^m_k(\epsilon^m_{rz,k} - r\epsilon^m_{z,k} - \epsilon_r(z^m_k + \epsilon^m_{z,k})) \\
& + \sum_{\ell=m+1}^{d}\left(\sum_{k=1}^{w}\hat{\alpha}^{\ell}_k(\epsilon^{\ell}_{rz,k} - r\epsilon^{\ell}_{z,k} - \epsilon_r(z^{\ell}_k + \epsilon^{\ell}_{z,k}))\right) \\
& + \ \epsilon_r\left[\sum_{k=1}^{q-1}\hat{\alpha}^m_k z^m_k + \sum_{\ell=0}^{m-1}\sum_{k=1}^{w}\hat{\alpha}^{\ell}_k z^{\ell}_k\right]
\end{aligned}$$

This is equivalent to separating out all the terms on the right hand side that are of the form $\hat{\alpha}_q^\ell \times$ (something) for all $\ell \in [d]$.

$$\begin{aligned}
\hat{\alpha}_q^m \left(\epsilon_r(z_q^m + \epsilon_{z,q}^m) - \epsilon_{rz,q}^m + r\epsilon_{z,q}^m\right) = \; & \epsilon_v - (r+\epsilon_r)\epsilon_u + \sum_{k=q+1}^{w} \hat{\alpha}_k^m(\epsilon_{rz,k}^m - r\epsilon_{z,k}^m - \epsilon_r(z_k^m + \epsilon_{z,k}^m)) \\
& + \sum_{\ell=m+1}^{d}\left(\sum_{k=1,k\neq q}^{w} \hat{\alpha}_k^\ell(\epsilon_{rz,k}^\ell - r\epsilon_{z,k}^\ell - \epsilon_r(z_k^\ell + \epsilon_{z,k}^\ell))\right) \\
& + \sum_{\ell=m+1}^{d} \hat{\alpha}_q^\ell \left(\epsilon_{rz,q}^\ell - r\epsilon_{z,q}^\ell - \epsilon_r(z_q^\ell + \epsilon_{z,q}^\ell)\right) \\
& + \epsilon_r\left[\sum_{k=1}^{q-1}\hat{\alpha}_k^m z_k^m + \sum_{\ell=0}^{m-1}\sum_{k=1,k\neq q}^{w}\hat{\alpha}_k^\ell z_k^\ell\right] + \epsilon_r\left[\sum_{\ell=0}^{m-1}\hat{\alpha}_q^\ell z_q^\ell\right]
\end{aligned}$$

Substituting $\hat{\alpha}_q^\ell = p_q^\ell + \alpha_q^0 \prod_{j=0}^{\ell}(\beta + \sum_{i=0}^{j}\epsilon_\beta^i)$ for all $\ell \in [d]$, we get

$$\begin{aligned}
& \alpha_q^0\left[\sum_{\ell=m}^{d}\left[\left(\prod_{j=0}^{\ell}(\beta + \sum_{i=0}^{j}\epsilon_\beta^i)\right)\left(\epsilon_r(z_q^\ell + \epsilon_{z,q}^\ell) - \epsilon_{rz,q}^\ell + r\epsilon_{z,q}^\ell\right)\right]\right] - \alpha_q^0\left[\sum_{\ell=0}^{m-1}\epsilon_r z_q^\ell\prod_{j=0}^{\ell}(\beta + \sum_{i=0}^{j}\epsilon_\beta^i)\right] \\
& = \epsilon_v - (r+\epsilon_r)\epsilon_u + \sum_{k=q+1}^{w}\hat{\alpha}_k^m(\epsilon_{rz,k}^m - r\epsilon_{z,k}^m - \epsilon_r(z_k^m + \epsilon_{z,k}^m)) \\
& + \sum_{\ell=m+1}^{d}\left(\sum_{k=1,k\neq q}^{w}\hat{\alpha}_k^\ell(\epsilon_{rz,k}^\ell - r\epsilon_{z,k}^\ell - \epsilon_r(z_k^\ell + \epsilon_{z,k}^\ell))\right) + \sum_{\ell=m}^{d} p_q^\ell\left(\epsilon_{rz,q}^\ell - r\epsilon_{z,q}^\ell - \epsilon_r(z_q^\ell + \epsilon_{z,q}^\ell)\right) \\
& + \epsilon_r\left[\sum_{k=0}^{q-1}p_k^m z_k^m + \sum_{\ell=0}^{m-1}\sum_{k=1,k\neq q}^{w}\hat{\alpha}_k^\ell z_k^\ell\right] + \epsilon_r\left[\sum_{\ell=0}^{m-1}p_q^\ell z_q^\ell\right] \qquad (4.3)
\end{aligned}$$

Left hand side of this equation can be re-written as

$$\begin{aligned}
\alpha_q^0 \Bigg(& r\epsilon_{z,q}^\ell\left[\sum_{\ell=m}^{d}\left(\prod_{j=0}^{\ell}(\beta + \sum_{i=0}^{j}\epsilon_\beta^i)\right)\right] + \left[\sum_{\ell=m}^{d}\left[\left(\prod_{j=0}^{\ell}(\beta + \sum_{i=0}^{j}\epsilon_\beta^i)\right)\left(\epsilon_r(z_q^\ell + \epsilon_{z,q}^\ell) - \epsilon_{rz,q}^\ell\right)\right]\right] \\
& - \left[\sum_{\ell=0}^{m-1}\epsilon_r z_q^\ell\prod_{j=0}^{\ell}(\beta + \sum_{i=0}^{j}\epsilon_\beta^i)\right]\Bigg)
\end{aligned}$$

Let the above term be equal to $\alpha_q^0 \cdot y$, where y is the term within $(\cdot)$.

Now, equation 4.3 holds if either of the following hold:

1. If $y \neq 0$ Since α_q^0 is uniformly distributed, the probability that in this case the equality in

equation 4.3 holds is $1/|\mathbb{F}|$.

2. Or if $y = 0$, then

$$r\epsilon_{z,q}^{\ell}\left[\sum_{\ell=m}^{d}\left(\prod_{j=0}^{\ell}(\beta+\sum_{i=0}^{j}\epsilon_{\beta}^{i})\right)\right] = -\left[\sum_{\ell=m}^{d}\left[\left(\prod_{j=0}^{\ell}(\beta+\sum_{i=0}^{j}\epsilon_{\beta}^{i})\right)\left(\epsilon_r(z_q^{\ell}+\epsilon_{z,q}^{\ell})-\epsilon_{rz,q}^{\ell}\right)\right]\right] + \left[\sum_{\ell=0}^{m-1}\epsilon_r z_q^{\ell}\prod_{j=0}^{\ell}(\beta+\sum_{i=0}^{j}\epsilon_{\beta}^{i})\right]$$

In this case, either $\epsilon_{z,q}^{\ell}\left[\sum_{\ell=m}^{d}\left(\prod_{j=0}^{\ell}(\beta+\sum_{i=0}^{j}\epsilon_{\beta}^{i})\right)\right] = 0$. Since this is a uni-variate polynomial in β with degree at most d, it has at most d roots. Since β was sampled uniformly, the probability that β is equal to one of these roots is $d/|\mathbb{F}|$. Or $\epsilon_{z,q}^{\ell}\left[\sum_{\ell=m}^{d}\left(\prod_{j=0}^{\ell}(\beta+\sum_{i=0}^{j}\epsilon_{\beta}^{i})\right)\right] \neq 0$. Since r is uniformly distributed in $\mathbb{F}$, the probability that in this case the equality in equation 4.3 holds is $1/|\mathbb{F}|$.

Hence, overall the probability that that the view generated by the simulator in Case 2 is distinguishable from the view in the real execution is at most

$$\frac{1}{|\mathbb{F}|}+\left(1-\frac{1}{|\mathbb{F}|}\right)\left(\frac{d}{|\mathbb{F}|}+\left(1-\frac{d}{|\mathbb{F}|}\right)\frac{1}{|\mathbb{F}|}\right) < \frac{d+1}{|\mathbb{F}|}$$

In both cases, the probability of equality is upper bounded by $\frac{(d+1)}{|\mathbb{F}|}$. Therefore, the protocol is secure, since if the adversary induces errors of the form ϵ_r^{ℓ} or $\epsilon_{z,k}^{\ell}$ or $\epsilon_{rz,k}^{\ell}$ or ϵ_u or ϵ_v, then the value T computed during verification will be zero with probability at most $\frac{(d+1)}{|\mathbb{F}|}$. In the case where $T \neq 0$, $f_{\mathsf{checkZero}}$ fails (in detection) with probability at most $\frac{1}{|\mathbb{F}|}$. Thus overall, the probability of distinguishing between the real and ideal world is at most $\frac{(d+2)}{|\mathbb{F}|}$. For reasonable-sized fields, this is negligible in the security parameter.

Operating over Smaller fields. This protocol works for fields that are large enough such that $\frac{(d+2)}{|\mathbb{F}|}$ is an acceptable probability of an adversary cheating. In cases where it might be desirable to instead work in a smaller field, we can use the same approach as used by Chida et al. [CGH+18]. In particular, instead of having a single randomized evaluation of the circuit w.r.t. r, we can generate shares for δ random values $r_1, \ldots, r_\delta$ (such that $(\frac{(d+2)}{|\mathbb{F}|})^{\delta}$ is negligible in the security parameter) and

run multiple randomized evaluations of the circuit and verification steps for each r_i. Since each r is independently sampled and their corresponding verification procedures are also independent, this will yield a cheating probability of at most $(\frac{(d+2)}{|\mathbb{F}|})^\delta$, as required.

□

4.7 Weakly Private Fluid MPC

In this section, we describe a linear-based Fluid MPC that achieves weak privacy against malicious R-adaptive adversaries. This is an adaptation of the protocol by Gennaro et el. [GRR98], which is an optimized version of the semi-honest BGW [BGW88] protocol in the fluid MPC setting. For simplicity, throughout this section, we assume that the number of clients and number of servers in each committee are n. While in most places it is easy to see how the protocol can be extended to support committees of different sizes, we add additional remarks wherever necessary.

4.7.1 Linear Protocols

In this section, we discuss the sub-protocols that are used in our protocol. Each of these sub-protocols is a *linear protocol* (see Definition 13). Instantiating the protocol from Definition 15 with these sub-protocols, we get our weakly private linear-based Fluid MPC protocol. Each of these linear protocols is described between parties: $\mathcal{P}^1 = \{P_1^1, \ldots, P_n^1\}$ and $\mathcal{P}^2 = \{P_1^2, \ldots, P_n^2\}$.

Linear Protocol for π_{rand}. This protocol outputs honestly computed shares of random values or $\perp$. Parties in $\mathcal{P}^1$ sample random values and secret share them amongst the parties in $\mathcal{P}^2$. The parties in $\mathcal{P}^2$ compute a sum of these shares to obtain shares of a random value. A formal description of the protocol is given in Figure 4.5.

Linear Protocol for π_{input}. This is a simple input sharing protocol where in the computation phase, the parties in $\mathcal{P}^1$ computes secret shares of their inputs and send them to the parties in $\mathcal{P}^2$ during the hand-off phase.

Protocol π_{rand}

Inputs: The parties do not have any inputs.
Protocol: The parties proceed as follows:

- **Computation Phase:** Each party $\{P_i^1\}$ (for $i \in [n]$) chooses a random element $u_i \in \mathbb{F}$. It runs $\mathsf{share}(u_i)$ to receive shares $\{u_{i,j}\}_{j\in[n]}$.
- **Hand-off Phase:** For each $i, j \in [n]$, P_i^1 sends $u_{i,j}$ to party P_j^2.
- **Output Phase:** Given shares $([u_1], \ldots [u_n])$, the parties in $\mathcal{P}^2$ compute and output

$$[r] = \sum_{i\in[n]} [u_i]$$

Figure 4.5: Fluid Sub-Protocol π_{rand}

Linear Protocol for π_{mult}. This is the multiplication protocol used in BGW [BGW88] adapted to our setting, where the input sharings $[x], [y]$ are held by the parties in $\mathcal{P}^1$ who want to securely compute and send shares $[x \cdot y]$ to the parties in $\mathcal{P}^2$. A formal description of this protocol is given in Figure 4.6. Note that in this protocol, the parties in $\mathcal{P}^1$ (and the ones in $\mathcal{P}^2$) do not communicate amongst themselves, their is only one round of interaction where all the parties in $\mathcal{P}^1$ send messages to all the parties in $\mathcal{P}^2$.

Protocol π_{mult}

Inputs: The parties in $\mathcal{P}^1$ hold shares $[x], [y]$.
Protocol: The parties proceed as follows:

- **Computation Phase:** The parties in $\mathcal{P}^1$ locally compute $\langle x \cdot y\rangle = [x] \cdot [y]$. Let xy_i be the resulting share held by P_i^1. Each P_i^1 (for $i \in [n]$) runs $\mathsf{share}(xy_i)$ on their share xy_i to receive shares $\{xy_{i,j}\}_{j\in[n]}$.
- **Handoff Phase:** For each $i, j \in [n]$, P_i^1 sends $xy_{i,j}$ to party P_j^2.
- **Output Phase:** Parties in $\mathcal{P}^2$ locally compute and output $[x \cdot y] = \sum_{i\in[2t+1]} c_i \cdot [xy_i]$, where each c_i is the Lagrange reconstruction coefficient for a degree $2t$ polynomial.

Figure 4.6: Fluid Sub-Protocol π_{mult}

Linear Protocol for π_{trans}. This is a protocol for secure transfer, where the parties in $\mathcal{P}^1$ hold shares of a value x and wish to securely re-share it amongst the parties in $\mathcal{P}^2$. A formal description of this

protocol is given in Figure 4.7.

Protocol π_{trans}

Inputs: The parties in $\mathcal{P}^1$ hold shares $[x]$.
Protocol: The parties proceed as follows:

- **Computation Phase:** Each P_i^1 (for $i \in [n]$) runs $\mathsf{share}(x_i)$ on their share x_i to receive shares $\{x_{i,j}\}_{j \in [n]}$.
- **Hand-off Phase:** For each $i, j \in [n]$, P_i^1 sends $x_{i,j}$ to party P_j^2.
- **Output Phase:** Parties in $\mathcal{P}^2$ locally compute and output $[x] = \sum_{i \in [t+1]} c_i \cdot [x_i]$, where each c_i is the Lagrange reconstruction coefficient for a degree t polynomial.

Figure 4.7: Fluid Sub-Protocol π_{trans}

4.7.2 Proof of Weak Privacy

In this section, we show that the linear-based Fluid MPC protocol described in Definition 15, when instantiated with the sub protocols in Sections 4.7.1 for n clients and $\vec{n}$ servers achieves weak privacy (see Definition 16) against a malicious R-adaptive adversary controlling at most $t < n/2$ servers in each epoch and at most $t < n/2$ clients. This protocol achieves maximal fluidity.

Lemma 12. *Let f be an n-input functionality and C be a layered arithmetic circuit representing f. Let n, t be positive integers such that $n \geq 2t + 1$. The protocol defined in Definition 15 instantiated with linear protocols from Section 4.7.1 is weakly private against a malicious R-adaptive adversary controlling at most t servers in each epoch and at most t clients,*

Proof. We begin by describing the simulator.

Simulator. Until the end of the computation phase of the first layer of the circuit, as and when the adversary corrupts these servers, for each newly corrupted server S_i^1, the simulator sends $(1, i)$ to the trusted functionality and does the following:

- **Input gates:** For each input gate G_j held by an honest client P_j, it samples a random share $z_{j,i}^0$ on behalf of that honest client and sends to the adversary.

– **Random input gates:** For each random input gate G^r_k, the simulator samples a random share $u_{k,j,i}$ on behalf of each honest client P_j and sends them to the adversary.

Execution Stage: For each epoch ($\ell \in [d]$), the simulator does the following. Since the servers are allowed to volunteer in as many epochs as they want, let $\hat{\mathcal{S}}^{\ell+1}$, where $|\hat{\mathcal{S}}^{\ell+1}| \leq t$ be corrupt servers in $\mathcal{S}^{\ell+1}$ that the adversary had already corrupted in some prior epoch that they were part of (we will call them pre-corrupted in the context of this epoch). In addition to these, the adversary is also allowed to adaptively corrupt more servers in $\mathcal{S}^{\ell+1}$ from the beginning of the hand-off phase of epoch ℓ, until the end of the computation phase of epoch $\ell+1$ as long as the total number of corruptions do not exceed t in the current or any prior epoch (we will call them newly corrupted in the context of this epoch). The simulator sends $\mathsf{continue}$ to the trusted functionality and proceeds as follows:

– **Corruption within the epoch:** For each pre-corrupted and newly corrupted server $P_i^{\ell+1}$, it sends $(\ell+1, i)$ to the trusted functionality. For each gate G^ℓ_k (for $k \in [w]$), the simulator samples a random share $z^\ell_{k,i,j}$, on behalf of each honest server in the set P^ℓ_j and sends them to the adversary.

– **Handling Retroactive effect:** For each newly corrupted server $P_i^{\ell+1}$, if it was part of the execution phase in any prior epoch, then the simulator does the following. It sends (ℓ', i') to the trusted functionality. For each $\ell' < \ell + 1$ that $P_i^{\ell+1}$ was a part of, let i' be its assigned position in that epoch. For each $k \in [w]$, the simulator samples a random value $z^{\ell'}_{k,i'}$ and computes an honest t-out-of-n secret sharing $[z^{\ell'}_{k,i'}]$ of this value that is consistent with the shares $\{z^{\ell'}_{k,i',j}\}_{j \in \mathsf{Adv} \cap \mathcal{S}^{\ell'+1}}$ sent by the simulator on behalf of this party to the corrupt parties in epoch ℓ'. It sends this value along with all the n shares to the adversary.

If at any point during the execution phase, the adversary aborts, then the simulator sends abort to the trusted functionality.

Indistinguishability Argument. Throughout the protocol, the messages sent by each server or client to the next set of servers are always a sharing of some value. Since the adversary only controls

at most t parties in each committee, by the privacy property of Shamir secret sharing with privacy threshold t, the distribution of messages received by the adversary from every honest client or server during each round of communication is indistinguishable from a uniformly sampled value and does not depend on the value the honest client or server shared. Therefore, it suffices for the simulator to send random values to the adversary on behalf of each honest server/client. Moreover, even while handling retroactive effect, the simulator can simply compute and send to the adversary, shares of a random value (say v), as long as they are consistent with the shares sent for the remaining corrupt parties. Recall that in the real world, this value v corresponds to the value obtained by locally multiplying or adding (depending on the gate) shares of the incoming wires values of that gate. To an adversary who corrupts at most t servers in every committee, these shares of the incoming wires values appear uniformly distributed. As a result, the value v also appears uniformly distributed. Finally, the list of corrupted servers is also determined identically in the real and ideal worlds, and hence the joint distribution of the list of corrupted servers and the view of the adversary in the real and ideal executions is indistinguishable. □

Remark. This protocol trivially extends to the setting where each server set consists of a different number of servers. In this setting, we allow up to $t_i < |\mathcal{S}^i|/2$ corruptions in server set $\mathcal{S}^i$ and for each retro-active corruption, the simulator computes t_i-out-of-n_i secret sharing instead of t-out-of-n.

Combining Lemma 12 with Theorem 2 and subsequently with Theorem 3, we get the following corollary.

Corollary 1. *There exists an information-theoretically secure Maximally-Fluid MPC with R-Adaptive Security (see Definition 11) for any $f \in P/Poly$.*

4.8 Implementation and Evaluation

We implement our protocol in C++, using the evaluation code written Chida et al. [CGH+18] as a starting point. Chida et. al. is a state of the art, honest-majority malicious security compiler with constant overhead in the static setting. Both the initial code base and our modification relies on

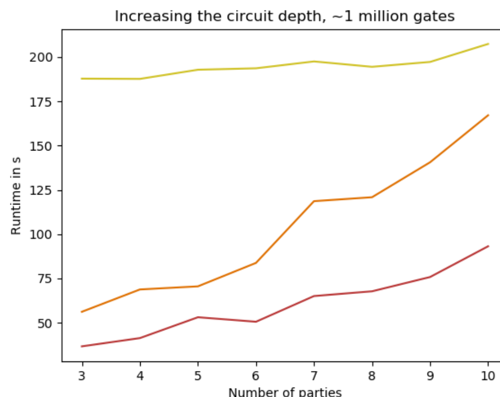


Figure 4.8: The computation phase runtimes of circuits with depths 10 (red), 100 (orange) and 1000 (yellow), but approximately equal numbers of multiplication gates.

the `libscapi` [Cry19] library to facilitate communication and evaluate field operations. `libscapi` supports a number of different fields, but we choose to execute all of our tests using the 61-bit Mersenne field. We note that the probability of detecting a malicious adversary with our compiler is proportional to the depth of the circuit. As such, for very deep circuits, the size of the field may need to be chosen accordingly. All communication was over unencrypted TCP point to point channels.

In our implementation, we incorporate a number of optimizations that are not included in our initial protocol description, that we omitted to streamline the intuition and analysis. In our formal description of the protocol, we introduce relay gates to signify transitioning data between committees. These relay gates also make explicit the need to re-share wire values connecting to gates deeper in the circuit than the immediate next layer. In our implementation, we chose not to alter the arithmetic circuit representation used by `libscapi` and instead keep track of where relay gates *would* be injected. To do this, we preprocess all wires in the circuit to count the number of times they are used in the circuit and decrement that value each time the wire is used as input to a gate being evaluated. Once this value reaches zero, it is no longer passed during the communication round. Importantly, our implementation, therefore, does not require circuits to be strictly layered.

In order to minimize the number of alpha values that need to be sent between committees, we add an additional preprocessing step to count the width of each layer of the circuit. Instead of sending a fixed number of alpha values at each layer, the parties only send a number of alphas equal

Configuration		Number of Parties								
Net Config	Width	3	4	5	6	7	8	9	10	20
LAN	100	0.389	0.458	0.516	0.550	0.686	0.758	0.990	1.036	3.171
LAN	1000	2.441	3.180	3.577	3.822	5.099	5.605	6.683	7.294	22.939
WAN	150	184.891	183.335	184.149	183.643	185.319	186.131	186.243	185.871	370.906
WAN	1500	186.823	187.683	189.532	189.905	195.937	192.087	195.443	200.885	1842.295

Table 4.1: Computation time for Fluid MPC, in milliseconds, per layer of the circuit.

to the maximum width of any future layer. While this optimization is insignificant in rectangular circuits, the savings can be considerable when circuits are more triangular in shape.

Because our implementation is intended to evaluate the efficiency of our protocol, we make the simplifying assumption that the parties are fixed for the duration of the protocol. While this might seem like a significant departure from the protocol described in Section 4.6, we note that switching between committees is not important for evaluating efficiency. The messages sent between parties and the computation performed do not change as a result of fixing the parties. Moreover, there are many possible ways to select which parties will be in each committee and we want our evaluation to be agnostic to these decisions. Finally, we keep the size of each committee fixed throughout the evaluation of each circuit.

4.8.1 Evaluation

In order to test our implementation, we needed to run it using varying number of parties and on circuits of various sizes. Because existing arithmetic circuit compilers infrastructure is lacking, we chose to generate randomized circuits instead of compiling specific functionalities. This randomized process allowed us to more carefully control the size and shape of the test circuits. Circuits were generated as follows: (1) A fixed number of inputs (1024 input wires for most of our test circuits) were randomly divided between the prescribed number of parties (2) The generator proceeds layer by layer for a prescribed number of layers. In each layer, it randomly selects a number of multiplication in $[\frac{w}{2}, 2]$ where w is the maximum width of any layer (another prescribed value). These gates are randomly connected to the output wires of the preceding layer. The generator also generates a random number of addition gates, subtraction gates, and scalar multiplication gates in $[\frac{w}{2}, 2]$, wiring them similarly. After this process, if there are any unconnected wires from the previous layer, the

generator inserts addition gates until all wires are connected. (3) Finally, the generator assigns the wires in the final layer as outputs to random parties. Using this method, we generate circuits of depth d that have between $\frac{wd}{2}$ and wd multiplication gates, and a similar number of addition gates.

We tested our protocol in both a LAN and WAN setting. The LAN configuration ran all parties on a single, large computer in our lab. The machine had 72 Intel Xeon E5 processors and 500GB of RAM. The WAN setup attempted to replicate the WAN deployment of [CGH+18]. We used AWS C4.large instances spread between North Virgina, Germany and India. Each party was run on a separate C4.large instance, even when the parties were located within the same zone. We report per-layer timing results for both our LAN deployment and WAN deployment in Table 4.1 (in WAN deployment, the communication time significantly dominates the time spent computing the gates. Note that the increase between 10 and 20 players is dramatic, as there are insufficient threads available on C4.large's for all parties to sync simultaneously.). Circuits for these tests were generated with the widths in the second column using techniques described above. Notably, the cost of doing wide area communication far outweighs the cost of local computation. The computation runtime of various depth circuits containing approximately 1 million gates is shown in Figure 4.8.

Bibliography

[ABF+17] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press. 16

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 805–817, Vienna, Austria, October 24–28, 2016. ACM Press. 16

[AHIV17] 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: 24th Conference on Computer and Communications Security*, pages 2087–2104, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. 8, 9

[B+14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014. 10

[BELO14] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. How to withstand mobile virus attacks, revisited. In Magnús M. Halldórsson and Shlomi Dolev,

editors, *33rd ACM Symposium Annual on Principles of Distributed Computing*, pages 293–302, Paris, France, July 15–18, 2014. Association for Computing Machinery. 14

[BGG+20] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? Cryptology ePrint Archive, Report 2020/464, 2020. https://eprint.iacr.org/2020/464. 14, 15, 16, 94, 96

[BGJK21] Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-C secure multiparty computation for highly repetitive circuits. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 663–693, Zagreb, Croatia, October 17–21, 2021. Springer, Heidelberg, Germany. 16

[BGW88] 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*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press. 2, 5, 13, 24, 82, 84, 108, 136, 137

[BHKL18] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. An end-to-end system for large scale p2p mpc-as-a-service and low-bandwidth mpc for weak participants. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 695–712, New York, NY, USA, 2018. ACM. 10

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press. 8

[BTH06] Zuzana Beerliová-Trubíniová and Martin Hirt. Efficient multi-party computation with dispute control. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryp-*

tography Conference, volume 3876 of *Lecture Notes in Computer Science*, pages 305–328, New York, NY, USA, March 4–7, 2006. Springer, Heidelberg, Germany. 87

[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. 34, 45

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (abstract) (informal contribution). In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, page 462, Santa Barbara, CA, USA, August 16–20, 1988. Springer, Heidelberg, Germany. 2, 5

[CDG+17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1825–1842, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. 8

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362, Cambridge, MA, USA, February 10–12, 2005. Springer, Heidelberg, Germany. 90, 99

[CGG+21] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid MPC: Secure multiparty computation with dynamic participants. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 94–123, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany. 16

[CGH+18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany. 3, 5, 7, 13, 14, 16, 32, 42, 44, 45, 58, 60, 61, 69, 77, 78, 79, 84, 85, 126, 135, 140, 143

[CH14] Michael R. Clark and Kenneth M. Hopkinson. Transferable multiparty computation with applications to the smart grid. *IEEE Trans. Inf. Forensics Secur.*, 9(9):1356–1366, 2014. 15

[CM19] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019. 12

[Cry19] Cryptobiu. cryptobiu/libscapi, May 2019. 14, 76, 77, 141

[DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 378–394, Santa Barbara, CA, USA, August 14–18, 2005. Springer, Heidelberg, Germany. 90, 99

[DI06] Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 501–520, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany. 5, 90, 99

[DIK+08] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 241–261, Santa Barbara, CA, USA, August 17–21, 2008. Springer, Heidelberg, Germany. 5, 38

[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. 5, 26, 30, 31, 38, 63, 64

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association. 3, 10, 94

[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. 3, 5, 24, 43, 57, 60, 73, 81

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany. 84, 89

[EOPY18] Karim Eldefrawy, Rafail Ostrovsky, Sunoo Park, and Moti Yung. Proactive secure multiparty computation with a dishonest majority. In Dario Catalano and Roberto De Prisco, editors, *SCN 18: 11th International Conference on Security in Communication Networks*, volume 11035 of *Lecture Notes in Computer Science*, pages 200–215, Amalfi, Italy, September 5–7, 2018. Springer, Heidelberg, Germany. 14

[FL19] Jun Furukawa and Yehuda Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Com-*

puter and Communications Security, pages 1557–1571. ACM Press, November 11–15, 2019. 3, 5, 7, 8, 16, 32, 42, 75, 77, 78, 79, 84, 85

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany. 8

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *24th Annual ACM Symposium on Theory of Computing*, pages 699–710, Victoria, BC, Canada, May 4–6, 1992. ACM Press. 5, 22, 25

[Gen16] Daniel Genkin. *Secure Computation in Hostile Environments*. PhD thesis, Technion - Israel Institute of Technology, 2016. 32, 63

[GFD09] Patrick Gallagher, Deputy Director Foreword, and Cita Furlani Director. Fips pub 186-3 federal information processing standards publication digital signature standard (dss), June 2009. U.S.Department of Commerce/National Institute of Standards and Technology. 40

[GHM+17a] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68, 2017. 12, 94

[GHM+17b] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. Cryptology ePrint Archive, Report 2017/454, 2017. https://eprint.iacr.org/2017/454. 99

[GHM+20] Craig Gentry, Shai Halevi, Bernardo Magri, Jesper Buus Nielsen, and Sophia Yakoubov. Random-index PIR with applications to large-scale secure MPC. Cryptology ePrint Archive, Report 2020/1248, 2020. https://eprint.iacr.org/2020/1248. 14

[GIP+14] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th Annual ACM Symposium on Theory of Computing*, pages 495–504, New York, NY, USA, May 31 – June 3, 2014. ACM Press. 7, 13, 16, 32, 84, 85, 107, 108, 109, 111, 112, 116, 117, 118, 121

[GIP15] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 721–741, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany. 5, 7, 26, 32, 38, 61, 62, 64, 84, 85

[GIW16] Daniel Genkin, Yuval Ishai, and Mor Weiss. Binary AMD circuits from secure multiparty computation. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part I*, volume 9985 of *Lecture Notes in Computer Science*, pages 336–366, Beijing, China, October 31 – November 3, 2016. Springer, Heidelberg, Germany. 85

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany. 90

[GKM+20] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. Cryptology ePrint Archive, Report 2020/504, 2020. `https://eprint.iacr.org/2020/504`. 15, 16

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security*

2016: 25th USENIX Security Symposium, pages 1069–1083, Austin, TX, USA, August 10–12, 2016. USENIX Association. 8

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th Annual ACM Symposium on Theory of Computing*, pages 291–304, Providence, RI, USA, May 6–8, 1985. ACM Press. 4

[GMW87] 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 Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press. 2

[Gol04] Oded Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004. 19

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In Brian A. Coan and Yehuda Afek, editors, *17th ACM Symposium Annual on Principles of Distributed Computing*, pages 101–111, Puerto Vallarta, Mexico, June 28 – July 2, 1998. Association for Computing Machinery. ix, 13, 82, 136

[GSB+16] Adria Gascon, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Secure linear regression on vertically partitioned datasets. Cryptology ePrint Archive, Report 2016/892, 2016. `https://eprint.iacr.org/2016/892`. 39

[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. 7, 32, 84, 85

[HJKY95] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In Don Coppersmith, editor, *Advances in Cryptology – CRYPTO'95*, volume 963 of *Lecture Notes in Computer Science*, pages 339–352, Santa Barbara, CA, USA, August 27–31, 1995. Springer, Heidelberg, Germany. 14

[HN06] Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 463–482, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany. 3

[IKHC14] Dai Ikarashi, Ryo Kikuchi, Koki Hamada, and Koji Chida. Actively private and correct MPC scheme in $t < n/2$ from passively secure schemes with small overhead. Cryptology ePrint Archive, Report 2014/304, 2014. https://eprint.iacr.org/2014/304. 16

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th Annual ACM Symposium on Theory of Computing*, pages 21–30, San Diego, CA, USA, June 11–13, 2007. ACM Press. 4, 8

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 525–537, Toronto, ON, Canada, October 15–19, 2018. ACM Press. 8

[LN17] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Con-*

ference on Computer and Communications Security, pages 259–276, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. 3, 7, 16

[MGC+16] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 112–127. IEEE, 2016. 40

[Mic17] Silvio Micali. Very simple and efficient byzantine agreement. In Christos H. Papadimitriou, editor, *ITCS 2017: 8th Innovations in Theoretical Computer Science Conference*, volume 4266, pages 6:1–6:1, Berkeley, CA, USA, January 9–11, 2017. LIPIcs. 12

[MR18] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 35–52, Toronto, ON, Canada, October 15–19, 2018. ACM Press. 39

[MRZ15] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 591–602, Denver, CO, USA, October 12–16, 2015. ACM Press. 16

[MZ17] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38, San Jose, CA, USA, May 22–26, 2017. IEEE Computer Society Press. 39

[MZW+19] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. CHURP: dynamic-committee proactive secret sharing. In *ACM Conference on Computer and Communications Security*, pages 2369–2386. ACM, 2019. 14

[Nak08] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. 2008. 3, 10, 40

[NIS02] Fips pub 180-2, secure hash standard (shs), 2002. U.S.Department of Commerce/National Institute of Standards and Technology. 40

[NV18] Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*, volume 10892 of *Lecture Notes in Computer Science*, pages 321–339, Leuven, Belgium, July 2–4, 2018. Springer, Heidelberg, Germany. 3, 5, 7, 13, 16, 32, 84, 85

[OY91] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In Luigi Logrippo, editor, *10th ACM Symposium Annual on Principles of Distributed Computing*, pages 51–59, Montreal, QC, Canada, August 19–21, 1991. Association for Computing Machinery. 14

[PS17] Rafael Pass and Elaine Shi. FruitChains: A fair blockchain. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *36th ACM Symposium Annual on Principles of Distributed Computing*, pages 315–324, Washington, DC, USA, July 25–27, 2017. Association for Computing Machinery. 90

[PSs17] Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 643–673, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany. 90

[RSG98] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998. 3

[Sha79a] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979. 21

[Sha79b] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979. 24

[WJS+19] Ryan Wails, Aaron Johnson, Daniel Starin, Arkady Yerukhimovich, and S. Dov Gordon. Stormy: Statistics in tor by measuring securely. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 615–632. ACM Press, November 11–15, 2019. 4, 78, 94

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press. 2