Communication Complexity of Information-Theoretic Multiparty Computation

Yifan Song

CMU-CS-22-118
July 2022

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Vipul Goyal, Chair
Bryan Parno
Elaine Shi
Yuval Ishai (Technion, Israel)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under award number CNS-1916939, the Defense Advanced Research Projects Agency under award number HR0011-20-2-0025, and the PNC Center. The author was also supported by Cylab Presidential Fellowships.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Information-Theoretic Cryptography, Multiparty Computation, Communication Complexity

Abstract

Since the notion of Multiparty Computation (MPC) was proposed three decades ago, a lot of research and effort has been done to improve the efficiency of MPC protocols. However, the inefficiency of the current state-of-the-art is still the major barrier that prevents MPC from being used more broadly. In this thesis, we study the communication complexity of information-theoretic MPC protocols.

Part I: Information Theoretic MPC with Honest Majority.

First, we study the honest majority setting, i.e., the number of corrupted parties is $t < n/2$, where n is the number of parties. We consider both the semi-honest security and malicious security (with abort).

In the setting of semi-honest security, we propose two improvements over the previously best-known result by Damgård and Nielsen (CRYPTO 2007). Our first protocol improves the communication complexity of the Damgård and Nielsen protocol from 6 field elements per party per gate to 4 elements. Our second protocol reduces the round complexity by a factor of 2 and achieves 4.5 field elements per party per gate.

As for malicious security, we show that it can be achieved with the same concrete efficiency as the semi-honest security. Previously best known results have a factor of 2 in the communication complexity when compiling a semi-honest protocol to a maliciously secure one. Our result closes the communication efficiency gap between semi-honest security and malicious security (with abort).

Part II: Information-Theoretic MPC with Dishonest Majority in the Preprocessing Model.

Then, we study the dishonest majority setting in the circuit-independent preprocessing model. We consider a sub-optimal corruption threshold where $t = (1-\epsilon) \cdot n$ for a constant ϵ.

In this thesis, we construct the first MPC protocols in the preprocessing model for dishonest majority which achieves sub-linear amount of preprocessing data and communication complexity per gate in the number of parties n. To achieve our results, we extend the use of packed secret sharing to the dishonest majority setting. For a constant fraction of corrupted parties (i.e. if 99 percent of the parties are corrupt), we achieve $O(1)$ field elements in both of the amount of preprocessing data and communication complexity per gate across all parties.

Acknowledgments

I feel fortunate and grateful for meeting many great people and receiving their help during my Ph.D. journey. First and foremost, I would like to thank my advisor Vipul Goyal for his invaluable guidance and generous support. It has been both a pleasure and an honor to be advised by Vipul. I would also like to thank my thesis committee members: Yuval Ishai, Bryan Parno, and Elaine Shi for their helpful and constructive comments on my thesis.

During my Ph.D. journey, I has been working with many excellent researchers. I would like to express thanks to my friends and collaborators: Abhiram Kothapalli, Hanjun Li, Yanyi Liu, Elisaweta Masserova, Rafail Ostrovsky, Antigoni Polychroniadou, Akshayaram Srinivasan, Chen-Da Liu-Zhang, and Chenzhi Zhu.

Finally, I has been enjoying my study and research at Carnegie Mellon University, which provides great working environment and generous funding support.

Contents

1 Introduction **1**
- 1.1 Setting . . . 2
- 1.2 Part I: Efficient Information-Theoretic MPC with Honest Majority . . . 3
- 1.3 Part II: Sharing Transformation and Applications to Dishonest Majority MPC with Packed Secret Sharing . . . 5

2 The Model: Secure Multiparty Computation **9**
- 2.1 Security Definition . . . 9
- 2.2 Hybrid Model and Preprocessing Model . . . 10
- 2.3 Client-server Model . . . 10

I Efficient Information-Theoretic MPC with Honest Majority **13**

3 Introduction for Part I **15**
- 3.1 Our Contributions . . . 16
- 3.2 Related Works . . . 16

4 Technical Overview **19**
- 4.1 Review: The Semi-Honest DN Protocol [28] . . . 19
- 4.2 Reducing the Communication Complexity via t-wise Independence . . . 20
- 4.3 Reducing the Number of Rounds via Beaver Triples . . . 21
- 4.4 Achieving Malicious Security . . . 25
  - 4.4.1 Review: Batch-wise Multiplication Verification . . . 26
  - 4.4.2 Extensions . . . 27
  - 4.4.3 Fast Verification for a Batch of Multiplication Tuples . . . 28
  - 4.4.4 Achieving Malicious Security for Our Two Semi-honest Protocols . . . 30

5 Preliminaries **33**
- 5.1 Shamir Secret Sharing Scheme . . . 33
- 5.2 Generating Random Sharings . . . 34
- 5.3 Generating Random Double Sharings . . . 36
- 5.4 Generating Random Coins . . . 38

6 Efficient Semi-Honest MPC Protocols **41**

6.1 Review of the Semi-Honest DN Protocol in [28] . . . 41
6.2 Reducing the Communication Complexity via t-wise Independence . . . 43
6.2.1 Our Observation . . . 43
6.2.2 ATLAS Multiplication Protocol . . . 44
6.2.3 Using $\mathcal{F}'_{\text{mult}}$ in the Semi-Honest DN protocol in [28] . . . 47
6.3 Reducing the Number of Rounds via Beaver Triples . . . 47
6.3.1 An Overview of Our Approach . . . 47
6.3.2 Improving the Original Multiplication Protocol in [28] . . . 49
6.3.3 Evaluating a Two-Layer Circuit . . . 52
6.3.4 Main Protocol . . . 53

7 Fast Verification for a Batch of Multiplication Tuples 59
7.1 Extension of the DN Multiplication Protocol . . . 60
7.2 Extension of the Batch-wise Multiplication Verification Technique . . . 63
7.3 Our Verification Protocol . . . 65
7.3.1 Step One: De-Linearization . . . 65
7.3.2 Step Two: Dimension-Reduction . . . 66
7.3.3 Step Three: Randomization . . . 68
7.3.4 Summary . . . 69
7.4 Compiling the Semi-Honest DN Protocol with Malicious Security . . . 73

8 Achieving Malicious Security 77
8.1 Achieving Malicious Security for the `t-wise` Variant . . . 77
8.2 Achieving Malicious Security for the `round-compression` Variant . . . 80

II Sharing Transformation and Applications to Dishonest Majority MPC with Packed Secret Sharing 87

9 Introduction for Part II 89
9.1 Our Contributions . . . 91
9.2 Related Works . . . 92

10 Technical Overview 97
10.1 Preparing Random Sharings for Different Linear Secret Sharing Schemes . . . 98
10.1.1 Starting Point - Preparing a Random Sharing for a Single Linear Secret Sharing Scheme . . . 99
10.1.2 Preparing Random Sharings for a Batch of Different Linear Secret Sharing Schemes . . . 99
10.2 Application: MPC via Packed Shamir Secret Sharing Schemes . . . 102
10.2.1 Network Routing . . . 104
10.2.2 Evaluating Multiplication Gates Using Packed Beaver Triples . . . 105
10.2.3 Summary . . . 106
10.2.4 An Alternative Approach for Network Routing . . . 106

10.2.5 Other Results . . . 110

11 Preparing Random Sharings for Different Arithmetic Secret Sharing Schemes 111
11.1 Arithmetic Secret Sharing Schemes . . . 111
11.2 Packed Shamir Secret Sharing Scheme . . . 113
11.3 Preparing Random Sharings for Different Arithmetic Secret Sharing Schemes . . 113
11.4 Instantiating Protocol RAND-SHARING via Packed Shamir Secret Sharing Scheme 118
11.5 Application of $\mathcal{F}_{\text{rand-sharing}}$ . . . 119

12 Semi-Honest Protocol 125
12.1 Circuit-Independent Preprocessing Phase . . . 125
12.2 Online Computation Phase . . . 125
12.2.1 Input Layer . . . 126
12.2.2 Network Routing . . . 126
12.2.3 Evaluating Addition Gates and Multiplication Gates . . . 128
12.2.4 Output Layer . . . 129
12.2.5 Main Protocol . . . 130

13 An Alternative Approach for Network Routing 135
13.1 Preliminaries: Hall's Marriage Theorem . . . 135
13.2 Network Routing using Hall's Marriage Theorem . . . 136

14 Maliciously Secure Protocol 141
14.1 Performing Sharing Transformation with Malicious Security . . . 141
14.1.1 Preparing Random Sharings for Sharing Transformations . . . 142
14.1.2 Performing Sharing Transformation . . . 146
14.2 Circuit-Independent Preprocessing Phase . . . 147
14.3 Online Computation Phase . . . 148
14.3.1 Input Layer . . . 148
14.3.2 Network Routing . . . 149
14.3.3 Evaluating Addition Gates and Multiplication Gates . . . 150
14.3.4 Output Layer . . . 151
14.4 Main Protocol . . . 156

15 Honest Majority MPC with Sublinear Online Communication 169
15.1 Preparing Random Sharings in Batch . . . 169
15.1.1 Preparing Verified Sharings. . . . 170
15.1.2 Converting to Random Sharings. . . . 171
15.2 Preprocessing for Honest Majority with Malicious Security . . . 175

Bibliography 179

List of Figures

5.1 **Functionality** $\mathcal{F}_{\text{rand}}$ . . . 34
5.2 **Protocol** RAND . . . 34
5.3 **Functionality** $\mathcal{F}_{\text{doubleRand}}$ . . . 36
5.4 **Protocol** DOUBLERAND . . . 37
5.5 **Functionality** $\mathcal{F}_{\text{coin}}$ . . . 38
5.6 **Protocol** COIN . . . 39

6.1 **Functionality** $\mathcal{F}_{\text{mult}}$ . . . 42
6.2 **Protocol** DN-MULT . . . 42
6.3 **Protocol** DN-MAIN . . . 43
6.4 **Protocol** EXPAND . . . 44
6.5 **Protocol** MULT . . . 44
6.6 **Functionality** $\mathcal{F}'_{\text{mult}}$ . . . 45
6.7 **Protocol** ATLAS-MULT . . . 45
6.8 **Functionality** $\mathcal{F}_{\text{zero}}$ . . . 50
6.9 **Protocol** ZERO . . . 50
6.10 **Protocol** EXPANDZERO . . . 52
6.11 **Protocol** IMPROVED-DN-MULT . . . 52
6.12 **Protocol** EVALUATE . . . 53
6.13 **Functionality** $\mathcal{F}_{\text{main}}$ . . . 54
6.14 **Protocol** MAIN-ROUND . . . 54

7.1 **Functionality** $\mathcal{F}_{\text{mult-mal}}$ . . . 59
7.2 **Functionality** $\mathcal{F}_{\text{multVerify}}$ . . . 60
7.3 **Functionality** $\mathcal{F}_{\text{extendMult}}$ . . . 61
7.4 **Protocol** EXTEND-MULT . . . 61
7.5 **Protocol** COMPRESS . . . 64
7.6 **Protocol** EXTEND-COMPRESS . . . 66
7.7 **Protocol** DE-LINEARIZATION . . . 67
7.8 **Protocol** DIMENSION-REDUCTION . . . 67
7.9 **Protocol** RANDOMIZATION . . . 69
7.10 **Protocol** MULTVERIFICATION . . . 70
7.11 **Functionality** $\mathcal{F}_{\text{main-mal}}$ . . . 73
7.12 **Protocol** DN-MAIN-MAL . . . 74

8.1 **Functionality** $\mathcal{F}'_{\text{mult-mal}}$ . . . 77

8.2 **Protocol** CHECKCONSISTENCY$(N, \{x^{(1)}, \ldots, x^{(N)}\})$ . . . 80
8.3 **Protocol** MAIN-ROUND-MAL . . . 82

11.1 **Functionality** $\mathcal{F}_{\text{rand-sharing}}(\Pi)$ . . . 114
11.2 **Functionality** $\mathcal{F}_{\text{randZero}}$ . . . 114
11.3 **Protocol** RAND-SHARING . . . 116
11.4 **Functionality** $\mathcal{F}_{\text{tran}}$ . . . 120
11.5 **Protocol** TRAN . . . 120

12.1 **Functionality** $\mathcal{F}_{\text{prep}}$ . . . 126
12.2 **Protocol** NETWORK . . . 128
12.3 **Protocol** PACKED-ADD . . . 129
12.4 **Protocol** PACKED-MULT . . . 129
12.5 **Protocol** PACKED-MAIN . . . 131

13.1 **Protocol** NETWORK-HALL-1 . . . 139
13.2 **Protocol** NETWORK-HALL-2 . . . 140

14.1 **Functionality** $\mathcal{F}_{\text{rand-sharing-mal}}$ . . . 142
14.2 **Protocol** RAND-SHARING-MAL . . . 143
14.3 **Protocol** TRAN-MAL . . . 146
14.4 **Functionality** $\mathcal{F}_{\text{prep-mal}}$ . . . 148
14.5 **Protocol** INPUT-MAL . . . 149
14.6 **Protocol** NETWORK-MAL . . . 150
14.7 **Protocol** PACKED-ADD-MAL . . . 151
14.8 **Protocol** PACKED-MULT-MAL . . . 152
14.9 **Functionality** $\mathcal{F}_{\text{com}}$ . . . 153
14.10 **Protocol** CHECK-CONSISTENCY . . . 154
14.11 **Protocol** CHECK-SECRET . . . 155
14.12 **Protocol** OUTPUT-MAL . . . 156
14.13 **Protocol** PACKED-MAIN-MAL . . . 158

15.1 **Protocol** VERSHARE(P_d, N') . . . 171
15.2 **Protocol** CONVERT . . . 172
15.3 **Protocol** BATCH-RAND(N) . . . 173

Chapter 1

Introduction

Consider the scenario where two millionaires are boasting about their wealth, and they are interested in learning who is richer *without revealing* their wealth to each other. A straightforward way would be asking a third person they trust to compare their wealth. However, what if such a third person does not exist? Can these two millionaires still learn the comparison result? This is a famous problem called "Yao's Millionaires' Problem" [65].

If we use x_1, x_2 to represent the wealth of these two millionaires respectively, they want to learn the output of a comparison function on x_1, x_2. This seemingly simple problem turns out to have a significant impact in the field of Cryptography. In [65], the solution proposed by Yao allows two parties to securely evaluate *any computable function*. Later on, Goldreich et al. [40] extended this result to the setting of multiple parties, which is known as secure Multiparty Computation (MPC).

Early feasibility solutions for MPC were proposed in [9, 40, 65], which showed that any function which can be computed, can be computed privately. Over the last three decades, a lot of research has been done to improve the efficiency of MPC protocols. Thanks to these efforts, MPC has rapidly moved from theory to practice.

Applications in the Real World — Breaking Barriers among Different Data Sets. The 21st century is the age of Big Data. Every place and every second, data is collected, analyzed, and transformed into knowledge and information which guide our lives. The larger amount of data we can gather and use, the higher chance we can obtain the desired knowledge and information. However, data is usually scattered among different organizations or companies. Sharing the data with others may either be restricted by laws or damage the interest of companies. There is an urgent need in providing a solution that maintains the privacy of each data set but allows data analyses among different data sets. Multiparty computation becomes a perfect tool as we discussed in the following concrete examples.

Consider various hospitals that may want to compute statistics of the success rate of a particular treatment or statistics related to side effects in patients taking a particular drug. Indeed, these kinds of statistics are very valuable in medical science research. However, due to the relevant laws and regulations, hospitals may not have the right to share patient data with anyone else (including each other) without the consent of every individual patient. Thus, it presents a significant obstacle to the hospitals towards doing any statistical analysis on the joint dataset.

However, using MPC, hospitals can run a protocol for the desired functionality with each dataset as private input. At the end of the protocol, they can learn statistics they want while protecting the privacy of the dataset held by each hospital.

Also, consider a bank that provides the loan service. To better assess the qualification of a customer, the bank may not only need the credit rating issued by the government, but also the information about house property, business, online shopping, income and expenses, and so on. Unfortunately, these pieces of information are usually held by different companies and organizations. These companies and organizations may be reluctant to share the information with others since it may damage their reputation in protecting the data privacy of their customers. Some information may even be forbidden to be shared by law. Relying on MPC, the bank and the related companies and organizations can together evaluate a proper function on their private data sets so that only the bank learns the qualification of the customer at the end of the protocol.

Other interesting examples include using MPC to realize distributed voting, private bidding, privacy-preserving machine learning, and so on.

Information-Theoretic Multiparty Computation and Its Communication Complexity. Although MPC brings us a strong primitive to handle privacy issues, the inefficiency of current constructions is the major barrier that prevents MPC from being used more broadly. There is an urgent need to improve the concrete efficiency of MPC protocols to make progress towards using them in the real world.

Usually, MPC protocols can be divided into two classes depending on whether cryptographic assumptions are required. Information-theoretic MPC protocol refer to those without any cryptographic assumptions. There are two major benefits of information-theoretic MPC protocols:

- First, information-theoretic MPC protocols are unconditionally secure. Namely, the security of these protocols does not rely on any hardness assumption and the protocol remains secure even if an adversary has *unlimited* computation power. In particular, such a protocol is not vulnerable to the power of quantum computing.
- Second, cryptographic primitives usually require heavy computation. By *not* using any cryptographic primitive, an information-theoretic MPC protocol typically has a lightweight local computation, often just a series of linear operations. As a result, the most efficient MPC protocols are in the information-theoretic MPC paradigm.

On the other hand, unlike the computation complexity, information-theoretic MPC protocols usually require a large amount of communication, which is proportional to the circuit size of the function. Since the cost of the local computation is small, *the main criterion for the efficiency of an information-theoretic MPC protocol is the amount of communication between every pair of parties*. This leads to our focus in this thesis — the communication complexity of information-theoretic MPC protocols.

1.1 Setting

In the setting of MPC, a set of n parties $\{P_1, P_2, \ldots, P_n\}$ together evaluate a common function f on their private inputs. The correctness of an MPC protocol requires that all parties should learn

the output of the common function at the end of the protocol when they follow the protocol. The security of an MPC protocol requires that any bounded number $t < n$ of corrupted parties together should not learn any information about honest parties' inputs (i.e., the inputs of the rest of $n - t$ parties) except what can be inferred from the function output, where the number t is referred to as the corruption threshold. In particular,

- If corrupted parties are required to follow the protocol, we say the MPC protocol achieves *semi-honest* security.
- If corrupted parties can arbitrarily deviate from the protocol, we say the MPC protocol achieves *malicious* security.

Usually, we transform the common function f to be computed to an arithmetic circuit over a finite field. The circuit supports addition and multiplication operations. Note that when the finite field is a binary field, we can use binary circuits to represent all computable functions. We assume that every pair of parties can send messages to each other via a secure and authenticated synchronized private channel.

1.2 Part I: Efficient Information-Theoretic MPC with Honest Majority

We first focus on the setting of honest majority where the number of corrupted parties $t = (n - 1)/2$ (i.e., $t < n/2$). The pioneering works [9, 23] gave the first positive results of information-theoretic MPC protocols with honest majority. In the meantime, the work [9] also gave a strong negative result that *information-theoretic MPC cannot exist without honest majority*. Indeed, for information-theoretic MPC protocols, requiring honest majority is the best we can hope.

Since [9, 23], a large number of works have focused on improving the efficiency of MPC protocols in various settings.

Semi-Honest Security. For over a decade, the most efficient MPC protocol with semi-honest security in the honest majority setting has been the protocol of Damgård and Nielsen [28], hereafter known as the DN protocol. Recall that the function we want to compute is transformed to an arithmetic circuit over a finite field. The DN protocol only needs to communicate 6 field elements per party per gate, and the overall communication complexity is $O(|C| \cdot n)$ field elements, where $|C|$ is the circuit size. Due to its simplicity and efficiency, many subsequent works have used the DN protocol to achieve a stronger security [10, 13, 17, 24, 38, 44, 60].

Despite the important role played by the DN protocol in the honest majority setting, any improvement to the basic protocol has been hard to come by. Our first result improves the DN protocol in the following two directions:

- We improve the basic DN protocol and reduces the communication complexity from 6 elements per party per gate to 4 elements. This gives the most communication-efficient semi-honest MPC protocol with honest majority.
- We then focus on the round complexity of the DN protocol. In the DN protocol, all gates in the same layer are evaluated in parallel. Therefore the number of rounds is linear in the

depth of the circuit. We show how to evaluate a two-layer circuit in parallel. This allows us to improve the concrete efficiency even further and reduce the number of rounds by a factor of 2. The achieved communication complexity is 4.5 elements per party per gate but halving the number of rounds.

Malicious Security. For malicious security, we consider the notion of security with abort, which allows a premature abort of the protocol. This means that a malicious adversary may prevent honest parties from learning the output of the common function, e.g., by aborting the protocol after learning the result. While it is not ideal, a malicious adversary still cannot learn any information about honest parties' inputs except what can be inferred from the function output. Furthermore, recent works [17, 44, 52] showed how to compile a maliciously secure-with-abort protocol to one with stronger security such as achieving fairness or guaranteed output delivery.

In [38], Genkin et al. showed that several semi-honest information theoretic MPC protocols, which include the protocols in [9, 28], are secure up to an additive attack in the presence of a fully malicious adversary. An additive attack means that the adversary is able to change the multiplication result by adding an arbitrary fixed value. Based on this observation, Genkin et al. [38] use a circuit which is resilient to an additive attack and give the first construction against a fully malicious adversary with communication complexity $O(|C| \cdot n)$ field elements, which matches the asymptotic communication complexity of the semi-honest DN protocol.

After that, a series of works [24, 56, 60] use different approaches to verify the correctness of the computation (i.e., to check whether the adversary launches additive attacks) with the concrete efficiency approaching to the semi-honest DN protocol. In particular, both works [24, 60] achieve the concrete communication efficiency of 12 field elements per multiplication gate, which is just 2x of the semi-honest DN protocol.

Despite all these improvements in the concrete efficiency, the question of whether the efficiency gap between malicious security (with abort) and semi-honest security is inherent in the honest majority setting still remains open.

Our second result is an efficient verification protocol with a sub-linear communication complexity in the circuit size. When combining the DN protocol and our efficient verification protocol, we can achieve the same concrete efficiency, i.e., 6 field elements per part per gate, thus closing the efficiency gap between malicious security (with abort) and semi-honest security. Furthermore, we utilize our efficient verification protocol and show how to compile our two semi-honest protocols with malicious security without affecting the concrete efficiency.

As a result, we obtain the following two theorems.

Theorem 1.1. *Let n be a positive integer and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n + 1$. For all arithmetic circuit C over $\mathbb{F}$, there is an information-theoretic n-party MPC protocol, which achieves malicious security (with abort) against a fully malicious adversary controlling $t < n/2$ corrupted parties, with communication complexity $O(|C| \cdot n + n^2 \cdot \kappa + n \cdot \kappa^2)$ elements, where κ is the security parameter and $|C|$ is the circuit size. Furthermore, the concrete efficiency is 4 elements per party per multiplication gate.*

Theorem 1.2. *Let n be a positive integer and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n + 1$. For all arithmetic circuit C over $\mathbb{F}$, there is an information-theoretic n-party MPC protocol, which achieves malicious security (with abort) against a fully malicious adversary controlling $t < n/2$*

corrupted parties, with communication complexity $O(|C| \cdot n + n^2 \cdot \kappa + n \cdot \kappa^2)$ elements, where κ is the security parameter and $|C|$ is the circuit size. Furthermore, the concrete efficiency is 4.5 elements per party per multiplication gate but halving the number of rounds (up to a constant number of rounds).

Our protocols are implemented in [45] and compared with the previously best-known results [24]. By combining improvements on both protocols, we achieve 2x speedup compared with [24]. Therefore, our protocols are the fastest known MPC protocols in this setting. The experiment result also shows the potential of using our protocols in the real world.

1.3 Part II: Sharing Transformation and Applications to Dishonest Majority MPC with Packed Secret Sharing

In the second part, we move our focus to the *dishonest majority* setting. To overcome the strong negative result by [9], we pin our hope on the circuit-independent *preprocessing model*. In the circuit-independent preprocessing model, all parties receive correlated randomness which is independent of the common function at the beginning of the protocol. The correlated randomness can be, for example, random OT pairs where one party holds two random values (r_0, r_1) and another party holds (b, r_b) where b is a uniform bit. One may think that there is a trusted third party that can prepare the correlated randomness locally and then distribute it to all parties. It can also be generated by using computational MPC protocols.

A celebrated work of Kilian [55] (improved later by [50]) shows that information-theoretic MPC protocols are possible to achieve with security-with-abort in the circuit-independent preprocessing model. In essence, *the circuit-independent preprocessing model allows us to push all the expensive cryptographic work into a preprocessing phase and take advantage of the high efficiency of information-theoretic protocols in the online phase.* The well-known SPDZ protocol [30] follows this strategy and becomes the most efficient protocol in the dishonest majority setting. In particular, the concrete efficiency of the SPDZ online protocol is only $3n$ field elements of preprocessing data and $4n$ field elements of communication per gate among all parties.

Dishonest Majority with Sub-optimal Threshold. We note that the SPDZ protocol [30] is secure against $t = n - 1$ corrupted parties, which is also known as the all-but-one corruption setting. However, this might be a too pessimistic assumption for real-world applications (in contrast to the honest majority setting which might be a too optimistic assumption). Therefore, we focus on the corruption threshold $t = (1 - \epsilon) \cdot n$ where ϵ is a constant.

In the honest majority setting with a sub-optimal threshold (i.e., $t = (1/2 - \epsilon) \cdot n$), a series of works [7, 29, 34, 39, 41, 46] considered to use the packed secret sharing technique [34] to reduce the communication complexity in the information-theoretic MPC protocols. In particular, the recent work [46] has shown that it is possible to achieve $O(1)$ field elements per gate among all parties when evaluating a single circuit. Note that it means that the cost per party is sub-linear in the number of participants. However, the cost of the SPDZ protocol is still $O(n)$ field elements per gate among all parties in the dishonest majority setting even with a sub-optimal threshold (i.e., $t = (1 - \epsilon) \cdot n$). This raises the following fundamental question:

"Is it possible to construct information-theoretic MPC protocols in the circuit-independent preprocessing model and dishonest majority setting with both the amount of preprocessing data and online communication complexity $O(1)$ field elements per gate among all parties?"

To answer this question, we first initiate the study of *sharing transformations* which allow us to perform arbitrary linear maps on the secrets of (packed) secret-sharing schemes.

Sharing Transformations. Consider two linear secret sharing schemes Σ and Σ' over a finite field $\mathbb{F}$. A set of n parties $\{P_1, P_2, \ldots, P_n\}$ start with holding a Σ-sharing $\boldsymbol{X}$. Here $\boldsymbol{X}$ could be the sharing of a single field element or a vector of field elements (e.g., as in packed secret sharing). The parties wish to compute a Σ'-sharing $\boldsymbol{Y}$ whose secret is a *linear map* of the secret of $\boldsymbol{X}$. Here a linear map means that each output secret is a linear combination of the input secrets (recall that the secret can be a vector in $\mathbb{F}$). We refer to this problem as *sharing transformation*.

Restricted cases of sharing transformations occur frequently in the construction of secure computation protocols based on secret sharing schemes. Generic approach of solving sharing transformations can achieve the optimal communication complexity (i.e., in the size of the sharings in Σ and Σ') when many sharing transformations of the same type are required. When we need to perform sharing transformations of different types, the generic approach becomes completely inefficient.

Our first result in this part is an efficient sharing transformation protocol which (1) can perform arbitrary linear maps, (2) is not restricted to the same type of sharing transformation, (3) can achieve the optimal communication complexity. We will show how our sharing transformation protocol can be used in constructing information-theoretic MPC protocols.

Information-Theoretic MPC Protocols with Packed Secret Sharings. The idea of the packed secret sharing technique [34] is to store multiple secrets within a single secret sharing. In this way, evaluating addition and multiplication gates over packed secret sharings computes the coordinate-wise addition and multiplication of the underlying secrets in parallel. Effectively, the amortized communication complexity per gate can be reduced by the packing parameter.

Following this approach, Franklin and Yung [34] showed how to evaluate many copies of the same circuit (with is also referred to as a SIMD circuit) with $O(1)$ field elements per gate among all parties when the corruption threshold is sub-optimal. After that, a series of works [7, 29, 39, 41] considered to use the packed secret sharing technique in evaluating a single circuit.

For a single circuit, the main difficulty is that the packed secret sharing we need for the current layer may contain secrets in multiple sharings from previous layers. The main difficulty is how to collect the secrets we need and compute a single packed secret sharing for them. This problem is referred to as network routing. Previous works solve this problem by either compiling the circuit to a SIMD circuit [29, 39], which leads to a $\log |C|$ overhead in the communication complexity, or restricting the type of circuits [7].

To solve the network routing, we proposed the idea of sparsely packed Shamir sharings where we store the values of different wires of the circuit in different slots (in contrast, the original packed secret sharing always use the same slots). In this way, we show that we can *locally* collect the secrets we need and compute a single packed secret sharing for them. However, this leads to the problem of evaluating addition and multiplication gates: the basic protocols for addition gates

and multiplication gates require the corresponding inputs to be correctly aligned up (i.e., the two inputs of the same gate should be stored in the same slot). We note that changing the slots of the secrets within a single sharing is a *sharing transformation*. Thus, we can use our efficient sharing transformation protocol to solve it. Note that the sharing transformation we need to perform can be different each time. The generic approach for sharing transformation, which can only achieve optimal communication complexity when performing many sharing transformations of the same type, is insufficient in our case.

Another contribution is that we manage to use the packed secret sharing technique in the *dishonest majority* setting.

To summarize, our second result in this part is an efficient information-theoretic MPC protocol in the dishonest majority setting which achieves $O(1)$ field elements in both the amount of communication complexity and the amount of preprocessing data with corruption threshold $t = (1 - \epsilon) \cdot n$, where ϵ is a constant. We have the following informal theorems.

Theorem 1.3 (Informal)**.** *For an arithmetic circuit C over a finite field $\mathbb{F}$ of size $|\mathbb{F}| \geq |C| + n$, there exists an information-theoretic MPC protocol in the preprocessing model which securely computes the arithmetic circuit C in the presence of a semi-honest adversary controlling up to t parties. The cost of the protocol is $O(|C| \cdot n^2/k^2)$ elements of preprocessing data, and $O(|C| \cdot n/k)$ elements of communication where $k = \frac{n-t+1}{2}$ is the packing parameter. For the case where $k = O(n)$, the achieved communication complexity in the online phase is $O(1)$ elements per gate.*

Theorem 1.4 (Informal)**.** *For an arithmetic circuit C over a finite field $\mathbb{F}$ of size $|\mathbb{F}| \geq 2^\kappa$, where κ is the security parameter, and for all $\frac{n-1}{3} \leq t \leq n - 1$, there exists an information-theoretic MPC protocol in the preprocessing model which securely computes the arithmetic circuit C in the presence of a fully malicious adversary controlling up to t parties. The cost of the protocol is $O(|C| \cdot n^2/k^2)$ elements of preprocessing data, and $O(|C| \cdot n/k)$ elements of communication where $k = \frac{n-t+1}{2}$ is the packing parameter. For the case where $k = O(n)$, the achieved communication complexity in the online phase is $O(1)$ elements per gate.*

Implications in the Honest Majority Setting. We observe that in the standard honest majority setting where the number of corrupted parties $t = (n-1)/2 \approx (1 - 1/2) \cdot n$, we can prepare the preprocessing data we need by using information-theoretic MPC protocols, and use our efficient protocol (in Theorem 1.4) with packed secret sharings in the online phase. In particular, the preprocessing data can be prepared by using the MPC protocols we constructed in Part I. As a result, we obtain an information-theoretic MPC protocol in the honest majority setting where the offline communication complexity is $O(|C| \cdot n)$ and the online communication complexity is $O(|C|)$. To the best of our knowledge, this is the first work that achieves sublinear online communication complexity in the number of parties in the information-theoretic setting with honest majority.

Chapter 2

The Model: Secure Multiparty Computation

We consider a set of parties $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ where each party can provide inputs, receive outputs, and participate in the computation. For every pair of parties, there exists a secure (private and authentic) synchronous channel so that they can directly send messages to each other. The communication complexity is measured by the number of bits X via private channels.

We focus on functions which can be represented as arithmetic circuits over a finite field $\mathbb{F}$ with input, addition, multiplication, and output gates[1]. We use κ to denote the security parameter and let $\mathbb{K}$ be an extension field of $\mathbb{F}$ (with $|\mathbb{K}| \geq 2^\kappa$). For simplicity, we use κ to denote the size of an element in $\mathbb{K}$. In this work, we assume that the number of parties n and the circuit size $|C|$ are bounded by polynomials of the security parameter κ.

2.1 Security Definition

Let $1 \leq t \leq n-1$ be an integer. Let $\mathcal{F}$ be a secure function evaluation functionality. An adversary $\mathcal{A}$ can corrupt at most t parties, provide inputs to corrupted parties, and receive all messages sent to corrupted parties. In this work, we consider both semi-honest adversaries and malicious adversaries.

- If $\mathcal{A}$ is semi-honest, then corrupted parties honestly follow the protocol.
- If $\mathcal{A}$ is fully malicious, then corrupted parties can deviate from the protocol arbitrarily.

Real-World Execution. In the real world, the adversary $\mathcal{A}$ controlling corrupted parties interacts with honest parties. At the end of the protocol, the output of the real-world execution includes the inputs and outputs of honest parties and the view of the adversary.

[1]In this work, we only focus on deterministic functions. A randomized function can be transformed into a deterministic function by taking as input an additional random tape from each party. The XOR of the input random tapes of all parties is used as the randomness of the randomized function.

Ideal-World Execution. In the ideal world, a simulator $\mathcal{S}$ simulates honest parties and interacts with the adversary $\mathcal{A}$. Furthermore, $\mathcal{S}$ has one-time access to $\mathcal{F}$, which includes providing inputs of corrupted parties to $\mathcal{F}$, receiving the outputs of corrupted parties, and sending instructions specified in $\mathcal{F}$ (e.g., asking $\mathcal{F}$ to abort). The output of the ideal-world execution includes the inputs and outputs of honest parties and the view of the adversary.

Semi-honest Security. We say that a protocol π computes $\mathcal{F}$ with perfect security if for all semi-honest adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that the distribution of the output of the real-world execution is *identical* to the distribution in the ideal-world execution.

Security-with-abort. We say that a protocol π securely computes $\mathcal{F}$ with abort if for all adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$, which is allowed to abort the protocol, such that the distribution of the output of the real-world execution is *statistically close* to the distribution in the ideal-world execution.

2.2 Hybrid Model and Preprocessing Model

The Hybrid Model. We follow [20] and use the hybrid model to prove security. In the hybrid model, all parties are given access to a trusted party (or alternatively, an ideal functionality) which computes a particular function for them. The modular sequential composition theorem from [20] shows that it is possible to replace the ideal functionality used in the construction by a secure protocol computing this function. When the ideal functionality is denoted by g, we say the construction works in the g-hybrid model.

The Preprocessing Model. In the (circuit-independent) preprocessing model, there is an ideal functionality which prepares circuit-independent correlated randomness before the computation. The preprocessing model can be viewed as a special case of the hybrid model. The cost of a protocol in the preprocessing model is measured by both the amount of communication via private channels and the amount of preprocessing data prepared by the ideal functionality [18, 25].

2.3 Client-server Model

To simplify the security proofs, sometimes we will consider the client-server model. In the client-server model, clients provide inputs to the functionality and receive outputs, and servers can participate in the computation but do not have inputs or get outputs. Each party may have different roles in the computation. Note that, if every party plays a single client and a single server, this corresponds to a protocol in the standard MPC model. Let c denote the number of clients and n denote the number of servers. For all clients and servers, we assume that every two of them are connected via a secure (private and authentic) synchronous channel so that they can directly send messages to each other. The communication complexity is measured in the same way as that in the standard MPC model.

Security in the Client-server Model. In the client-server model, an adversary $\mathcal{A}$ can corrupt at most c clients and t servers, provide inputs to corrupted clients, and receive all messages sent to corrupted clients and servers. The security is defined similarly to the standard MPC model.

Benefits of the Client-server Model. In our construction, the clients only participate in the input phase and the output phase. The main computation is conducted by the servers. For simplicity, we use $\{P_1, \ldots, P_n\}$ to denote the n servers, and refer to the servers as parties. Let $\mathcal{C}orr$ denote the set of all corrupted parties and $\mathcal{H}$ denote the set of all honest parties. One benefit of the client-server model is that it is sufficient to only consider maximum adversaries, i.e., adversaries which corrupt exactly t parties. At a high level, for an adversary $\mathcal{A}$ which controls $t' < t$ parties, we may construct another adversary $\mathcal{A}'$ which controls additional $t - t'$ parties and behaves as follows:

- For a party corrupted by $\mathcal{A}$, $\mathcal{A}'$ follows the instructions of $\mathcal{A}$. This is achieved by passing messages between this party and other $n - t'$ honest parties.
- For a party which is not corrupted by $\mathcal{A}$, but controlled by $\mathcal{A}'$, $\mathcal{A}'$ honestly follows the protocol.

Note that, if a protocol is secure against $\mathcal{A}'$, then this protocol is also secure against $\mathcal{A}$ since the additional $t - t'$ parties controlled by $\mathcal{A}'$ honestly follow the protocol in both cases. Thus, we only need to focus on $\mathcal{A}'$ instead of $\mathcal{A}$. Note that in the regular model, each honest party may have input. The same argument does not hold since the input of honest parties controlled by $\mathcal{A}'$ may be compromised.

Part I

Efficient Information-Theoretic MPC with Honest Majority

Chapter 3

Introduction for Part I

Secure Multi-Party Computation (MPC) allows $n \geq 2$ parties to compute a function on privately held inputs, such that the desired output is correctly computed and is the only new information released. This should hold even if t out of n parties have been corrupted by a semi-honest or malicious adversary. Since its introduction in the 1980s [40, 65], a lot of research has been done to improve the efficiency of MPC protocols. Thanks to these efforts, MPC has rapidly moved from theory to practice.

In this work, our focus is on honest majority protocols in the presence of a malicious adversary. We note that the fastest known implementations of MPC have come in the honest majority setting, which does not necessarily require public key operations. For example, the recent work of Chida et al. [24] showed that their secure-with-abort protocol can evaluate 1 million multiplication gates within 1 second for up to 7 parties, 4 seconds for 50 parties, and 8 seconds for 110 parties. Another attractive feature of the honest majority setting is that it allows one to achieve the stronger properties of fairness and guaranteed output delivery which are otherwise impossible with dishonest majority.

For over a decade, the most efficient MPC protocol with semi-honest security in the honest majority setting has been the protocol of Damgård and Nielsen [28], hereafter known as the DN protocol. By using the Shamir secret sharing scheme [64], addition gates can be evaluated without any communication. To evaluate a multiplication gate, each party only needs to communicate 6 field elements. In the computational setting, the communication complexity can be reduced to 3 field elements by using pseudo-random generators [60] (improved further to 1.5 elements by Boneh et al. [13] for a constant number of parties). Due to its simplicity and efficiency, many subsequent works have used the DN protocol to achieve security-with-abort [13, 17, 24, 38, 44, 60] or guaranteed output delivery [10, 44].

Despite the important role played by the DN protocol in the honest majority setting, any improvement to the basic protocol has been hard to come by unless one resorts to other approaches using computational assumptions.

In the setting of malicious security, the recent breakthrough [24, 60] showed that achieving security-with-abort requires only twice the cost of achieving semi-honest security compared with the DN protocol [28]. In the setting of $1/3$ corruption threshold, a recent beautiful work of Furukawa and Lindell [35] presented a construction which achieves the same communication cost as the DN protocol. When considering a 3-party computation for a binary circuit, a recent

work [4] presented a construction where each AND gate only requires 7 bits per party. As a result, over a billion AND gates could be processed within one second.

Despite all these improvements in the concrete efficiency, the question of whether the efficiency gap between malicious security (with abort) and semi-honest security is inherent in the honest majority setting still remains open. In this part, we ask the following natural question:

"Is it possible to achieve malicious security-with-abort with the same concrete cost *as the best-known semi-honest MPC protocol?"*

3.1 Our Contributions

We propose ATLAS, an unconditionally secure MPC protocol in the honest majority setting with reduced communication complexity over the celebrated DN protocol even in the honest but curious setting, as well as malicious setting. Our protocol ATLAS enjoys the following efficiency improvements over the DN protocol:

- We improve the basic DN protocol leading to a communication complexity of 4 field elements per multiplication gate per party. Our results are in the information-theoretic setting assuming a majority of the parties are honest and the adversary is semi-honest. This leads to the most communication-efficient semi-honest MPC protocol with honest majority.
- Next, we focus on the round complexity of the DN protocol. Instead of evaluating multiplication gates of the same layer in parallel, we show how to evaluate all multiplication gates in a two-layer circuit in parallel. This allows us to improve the concrete efficiency even further and reduce the number of rounds by a factor of 2. The achieved amortized communication cost per multiplication gate in this setting is 4.5 field elements per party but halving the number of rounds.

To achieve malicious security, we design an efficient verification protocol for multiplications building on the techniques in [13], which achieves a sub-linear communication complexity in the circuit size. As a result, we can make our two protocols secure against malicious adversaries without affecting the concrete efficiency.

The security of our construction does not depend upon the field size. One can use a field with size as low as $n+1$ where n is the number of parties. On the other hand, the concrete efficiency of both constructions from [24, 60] suffers from having a large field size. An alternative presented in [24] is to use a small field but then the verification must be done several times to reach the desired security parameter. This however would increase the number of field elements per multiplication gate several times. Another option presented in [60] allows one to reduce the field size without substantially increasing the number of fields elements per gate. However, the field size must still be at least as large as the circuit size and also depends upon the security parameter (and, e.g., cannot be a constant).

3.2 Related Works

In this section, we compare our result with several related constructions in both techniques and the efficiency. In the following, let $|C|$ denote the size of the circuit, κ denote the security

parameter, and n denote the number of parties participating in the computation.

Achieving Malicious Security with abort. In [28], Damgård and Nielsen introduce the best-known semi-honest protocol, which we refer to as the DN protocol. The communication complexity of the DN protocol is $O(Cn\phi)$ bits. The concrete efficiency is 6 field elements per multiplication gate (per party). In [38], Genkin, et al. show that the DN protocol is secure up to an additive attack when running in the fully malicious setting. Based on this observation, a secure-with-abort MPC protocol can be constructed by combining the DN protocol and a circuit which is resilient to an additive attack (referred to as an AMD circuit). As a result, Genkin, et al. [38] give the first construction against a fully malicious adversary with communication complexity $O(|C| \cdot n)$ field elements (for a large enough field), which matches the asymptotic communication complexity of the DN protocol.

The construction in [24] also relies on the theorem showed in [38]. The idea is to check whether the adversary launches an additive attack. In the beginning, all parties compute a random secret sharing of the value r. For each wire w with the value x associated with it, all parties will compute two secret sharings of the secret values x and $r \cdot x$ respectively. Here $r \cdot x$ can be seen as a secure MAC of x when the only possible attack is an additive attack. In this way, the protocol requires two operations per multiplication gate. The asymptotic communication complexity is $O(|C| \cdot n)$ field elements (for a large enough field) and the concrete efficiency is reduced to 12 field elements per multiplication gate.

An interesting observation is that the theorem showed in [38] implies that the DN protocol provides perfect privacy of honest parties (before the output phase) in the presence of a fully malicious adversary. To achieve security with abort, the only task is to check the correctness of the computation before the output phase. This observation has been used in [56, 60]. In particular, the construction in [60] achieves the same concrete efficiency as [24] by using the batch-wise multiplication verification technique in [10], i.e., 12 field elements per multiplication gate. Our construction also relies on this observation. Therefore, the main task is to efficiently verify a batch of multiplications such that the communication complexity is sublinear in the number of parties.

In [13], Boneh, et al. introduce a very powerful tool to achieve this task when the number of parties is restricted to be a constant. Our result is obtained by instantiating this technique with a different secret sharing scheme, which allows us to overcome this restriction so that it works for any (polynomial) number of parties. Furthermore, we simplify this technique by avoiding the use of a robust secret sharing scheme and a verifiable secret sharing scheme, which are required in [13]. Our protocol additionally makes a simple optimization to the DN protocol, which brings down the cost from 6 field elements per multiplication gate to 5.5 field elements. More details about the comparison for techniques can be found in the last paragraph of Section 4.4.3. A subsequent work [45] implements our construction and shows that the performance beats the previously best-known implementation result [24] in this setting.

A concurrent work [17] also follows [13] and gives a similar verification protocol for multiplications. As a result, they give an MPC protocol with 1.5 field elements per multiplication gate per party. However, their efficiency relies on the use of pseudo-random generators and is restricted to a constant number of parties.

Other Related Works. The notion of MPC was first introduced in [40, 65] in 1980s. Feasibility results for MPC were obtained by [22, 40, 65] under cryptographic assumptions, and by [9, 23] in the information-theoretic setting. Subsequently, a large number of works have focused on improving the efficiency of MPC protocols in various settings.

A series of works focus on improving the communication efficiency of MPC with guaranteed output delivery in the settings with different thresholds on the number of corrupted parties. In the setting of honest majority setting, assuming the existence of a broadcast channel, the works [10, 44] have shown that guaranteed output delivery can be achieved efficiently. In the setting where $t < n/3$, a rich line of works [8, 28, 43, 48, 49] have focused on improving the asymptotic communication complexity in this setting.

A rich line of works have also focused on the performance of MPC in practice for two parties [57, 59], or three parties [4, 36].

Chapter 4

Technical Overview

In this part, we consider the standard honest majority setting, i.e., the number of corrupted parties $t = (n-1)/2$. We give an overview of our techniques in this chapter. Our construction is based on the standard Shamir Secret Sharing Scheme [64]. We will use $[x]_d$ to denote a degree-d Shamir sharing, or a $(d+1)$-out-of-n Shamir sharing. It requires at least $d+1$ shares to reconstruct the secret and any d shares do not leak any information about the secret. We focus on finite fields of size at least $n+1$ to allow the existence of the Shamir secret sharing scheme. In the following, we use $\mathbb{F}$ to denote a finite field of size $|\mathbb{F}| \geq n+1$ and C to denote an arithmetic circuit over $\mathbb{F}$ that all parties want to evaluate.

We start with the semi-honest security, and then we will discuss how to achieve malicious security with the same concrete efficiency as the semi-honest protocols.

4.1 Review: The Semi-Honest DN Protocol [28]

In the DN protocol [28], all parties compute a degree-t Shamir sharing for each wire. With more details, all parties run the following three steps to evaluate an arithmetic circuit C over the finite field $\mathbb{F}$.

1. All parties start with sharing their inputs to other parties by using degree-t Shamir sharings over $\mathbb{F}$.
2. All parties evaluate the circuit C layer by layer. For each layer, all parties evaluate addition gates and multiplication gates as follows:
   - For each addition gate, given the two input sharings, all parties compute the output sharing by locally adding up their shares. This step relies on the linear homomorphism of the Shamir secret sharing scheme.
   - For each multiplication gate, given the two input sharings, all parties use the multiplication protocol in [28] (hereafter referred to as the DN multiplication protocol) to compute a degree-t Shamir sharing of the multiplication result.
3. After evaluating the whole circuit, all parties together reconstruct the output sharings.

Our idea is to reuse the correlated-randomness required in the DN multiplication protocol. We first review the DN multiplication protocol.

Review of the DN Multiplication Protocol. To evaluate a multiplication gate, all parties first need to prepare a pair of random sharings $([r]_t, [r]_{2t})$ of the same secret r, where the first sharing is a degree-t Shamir sharing and the second sharing is a degree-$2t$ Shamir sharing. Such a pair of sharings is referred to as a pair of double sharings. In [28], preparing a pair of random double sharings requires to communicate 4 elements per party.

For a multiplication gate, suppose the input sharings are denoted by $[x]_t, [y]_t$. To compute $[z]_t := [x \cdot y]_t$, a pair of random double sharings $([r]_t, [r]_{2t})$ is consumed. All parties first agree on a special party $P_{\texttt{king}}$. Then, all parties run the following steps:

1. All parties locally compute $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[e]_{2t}$ and reconstructs the secret e. Then $P_{\texttt{king}}$ sends the value e to all other parties.
3. After receiving e from $P_{\texttt{king}}$, all parties locally compute $[z]_t := e - [r]_t$.

Correctness follows from the properties of the Shamir secret sharing scheme. Note that each party needs to send an element to $P_{\texttt{king}}$, and $P_{\texttt{king}}$ needs to send an element to each party. The communication complexity of this protocol is 2 elements per party. Including the communication cost for preparing double sharings, the overall cost per multiplication gate is 6 elements per party.

4.2 Reducing the Communication Complexity via t-wise Independence

Starting Point. We observe that in the second step of the DN multiplication protocol, $P_{\texttt{king}}$ can alternatively distribute a degree-t Shamir sharing $[e]_t$. Then in the last step, all parties can still compute $[z]_t := [e]_t - [r]_t$. We note that, when $P_{\texttt{king}}$ is an honest party, the corrupted parties only receive several random elements from $P_{\texttt{king}}$ if $[e]_t$ is a random degree-t Shamir sharing. In particular, it holds even if the corrupted parties know the whole sharings $[r]_t$ and $[r]_{2t}$. This is because the corrupted parties only receive t shares of a random degree-t sharing $[e]_t$ from $P_{\texttt{king}}$, which are uniformly random and independent of the secret. Therefore for an honest $P_{\texttt{king}}$, we do not need the double sharings to be uniformly random at all. While for a corrupted $P_{\texttt{king}}$, we still need to use random double sharings, we can split the tasks of handling multiplication gates as $P_{\texttt{king}}$ to all parties. In this way, at least half of multiplication gates are handled by honest $P_{\texttt{king}}$'s. We show that it allows us to reduce the cost of preparing double sharings by a factor of 2.

Relying on t-wise Independence. Suppose we have n multiplication gates and we let each party behave as $P_{\texttt{king}}$ for 1 multiplication gate. When $P_{\texttt{king}}$ is a corrupted party, we still need to use a pair of random double sharings to protect the secrecy of the result. If $P_{\texttt{king}}$ is an honest party, as argued above, the double sharings do not need to be random.

Our idea is to generate n pairs of double sharings such that any t pairs of them are independent and uniformly random. This guarantees that the double sharings used for multiplication gates handled by corrupted parties are uniformly random, which ensures the security of the MPC protocol. On the other hand, given these double sharings, the other double sharings used for multiplication gates handled by honest parties can be fixed and determined. It means that we

only need to prepare t pairs of random and independent double sharings for n multiplication gates.

To this end, all parties agree on a fixed Vandermonde matrix of size $n \times t$, denoted by $\boldsymbol{M}$. The main property of $\boldsymbol{M}$ is that any $t \times t$ sub-matrix of $\boldsymbol{M}$ is invertible. Since the Shamir secret sharing scheme is a linear homomorphism, a linear combination of several pairs of double sharings is still a pair of double sharings. All parties first prepare t pairs of random double sharings using the protocol in [28], denoted by

$$([r^{(1)}]_t, [r^{(1)}]_{2t}), \ldots, ([r^{(t)}]_t, [r^{(t)}]_{2t}).$$

Then, we expand these t pairs of double sharings to n pairs by computing

$$\begin{aligned} ([\tilde{r}^{(1)}]_t, \ldots, [\tilde{r}^{(n)}]_t)^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_t, \ldots, [r^{(t)}]_t)^{\mathrm{T}} \\ ([\tilde{r}^{(1)}]_{2t}, \ldots, [\tilde{r}^{(n)}]_{2t})^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_{2t}, \ldots, [r^{(t)}]_{2t})^{\mathrm{T}}. \end{aligned}$$

We point out that this expansion can be done locally without interaction. Note that for all $i \in [n]$, $([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})$ is a pair of double sharings. Let $\mathcal{C}orr$ denote the set of corrupted parties. According to the property of $\boldsymbol{M}$, there is a one-to-one map from $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})\}_{i\in\mathcal{C}orr}$ to $\{([r^{(i)}]_t, [r^{(i)}]_{2t})\}_{i\in[t]}$. Since the input double sharings are independent and uniformly random, we conclude that the double sharings in $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})\}_{i\in\mathcal{C}orr}$ are independent and uniformly random.

When $([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})$ is used to evaluate a multiplication gate, we require the party P_i to act as $P_{\texttt{king}}$. In this way, the multiplication gates handled by corrupted parties will use double sharings in $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})\}_{i\in\mathcal{C}orr}$, which are independent and uniformly random. We are able to show that the security still holds.

Concrete Efficiency of Our Improved Multiplication Protocol. Recall that in [28], preparing a pair of random double sharings requires the communication of 4 elements per party. Relying on t-wise independence, we only need to prepare t pairs of random double sharings for n multiplications. Thus, the amortized communication cost per pair of double sharings is $4 \cdot t/n \approx 2$ elements per party. Including the communication cost of the multiplication protocol in [28], which is 2 elements per party, the overall cost per multiplication is 4 elements per party. As a result, we have the following theorem:

Theorem 4.1. *Let n be a positive integer and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n + 1$. For all arithmetic circuit C over $\mathbb{F}$, there is an information-theoretic n-party MPC protocol, which achieves semi-honest security against a semi-honest adversary controlling $t < n/2$ corrupted parties, with communication complexity $O(|C| \cdot n + n^2)$ elements, where $|C|$ is the circuit size. Furthermore, the concrete efficiency is 4 elements per party per multiplication gate.*

4.3 Reducing the Number of Rounds via Beaver Triples

In the DN protocol [28], multiplication gates in the same layer of the circuit are evaluated in parallel. Therefore, the number of rounds is linear in the depth of the circuit. To further improve the concrete efficiency, we pay attention to the round complexity. We note that the question of

obtaining information-theoretic constant round protocols for a general circuit has been opened for many years. In particular, it has been shown in [31] that the dependency on the depth in the round complexity is inherent for the DN protocol. Given this, we managed to reduce the number of rounds by a factor of 2 while maintaining the communication efficiency.

To this end, we first consider a two-layer circuit and try to evaluate all multiplication gates in parallel.

Starting Point. For a two-layer circuit, an input sharing of a multiplication gate in the second layer may come from three places:

- This sharing is an input sharing of the circuit.
- This sharing is an output sharing of an addition gate in the first layer.
- This sharing is an output sharing of a multiplication gate in the first layer.

Note that an addition gate can be evaluated without interaction. Therefore for the first two cases, all parties can locally compute this sharing. However, for the third case, communication is required to evaluate this multiplication gate in the first layer. Therefore, the question becomes how to evaluate multiplication gates in the second layer *without learning the output sharings of multiplication gates in the first layer*.

We recall the technique of Beaver triple [6]. A Beaver triple consists of three degree-t Shamir sharings $([a]_t, [b]_t, [c]_t)$, where a, b are random field elements and $c = a \cdot b$. Usually, a Beaver triple is used to transform one multiplication to two reconstructions. Concretely, given two sharings $[x]_t, [y]_t$, suppose we want to compute $[z]_t$ such that $z = x \cdot y$. Since

$$\begin{aligned} z &= x \cdot y \\ &= (x + a - a) \cdot (y + b - b) \\ &= (x + a) \cdot (y + b) - (x + a) \cdot b - (y + b) \cdot a + a \cdot b, \end{aligned}$$

we can compute

$$[z]_t := (x + a) \cdot (y + b) - (x + a) \cdot [b]_t - (y + b) \cdot [a]_t + [c]_t.$$

Therefore, the task of computing $[z]_t$ becomes to reconstructing two degree-t Shamir sharings $[x]_t + [a]_t$ and $[y]_t + [b]_t$. Observe that, if we set $u = x + a$ and $v = y + b$, the above equation allows us to locally compute a degree-t Shamir sharing of $z := (u - a) \cdot (v - b)$ using a Beaver triple $([a]_t, [b]_t, [c]_t)$ once u and v are publicly known.

Beaver-triple Friendly Form. We say a sharing is in the *Beaver-triple friendly form*, if it can be written as $u - [a]_t$, where u is a public element and $[a]_t$ is a degree-t Shamir sharing. Now suppose for each multiplication gate in the second layer, the input sharings are in the Beaver-triple friendly form, say $u - [a]_t$ and $v - [b]_t$. Given the Beaver triple $([a]_t, [b]_t, [c]_t)$, one can *non-interactively* compute the output sharing of this gate by

$$[z]_t := u \cdot v - u \cdot [b]_t - v \cdot [a]_t + [c]_t.$$

Note that the Beaver triple $([a]_t, [b]_t, [c]_t)$ can be prepared without learning u, v. Therefore, if for each multiplication gate in the second layer, the input sharings are in the Beaver-triple friendly

form $u - [a]_t, v - [b]_t$, and $[a]_t, [b]_t$ are learnt *before evaluating the first layer*, we can prepare the Beaver triple $([a]_t, [b]_t, [c]_t)$ without evaluating the first layer, and then non-interactively evaluate multiplication gates in the second layer after learning u, v from the first layer.

Of course, the question remains: since the input sharings of the second layer come from the output sharings of the first layer, how do we ensure that the output sharings of the first layer are in the Beaver-triple friendly form?

Evaluating a Two-Layer Circuit. We observe that the original DN multiplication protocol in [28] satisfies our requirement! Concretely, to evaluate a multiplication gate with input sharings $[x]_t, [y]_t$, all parties need to first prepare a pair of random double sharings $([r]_t, [r]_{2t})$. In the last step of the DN multiplication protocol, $P_{\texttt{king}}$ sends the reconstruction result of $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$ to all parties, and all parties can compute the degree-t Shamir sharing $[z]_t := e - [r]_t$. In particular, the output sharing is in the Beaver-triple friendly form, and the sharing $[r]_t$ is prepared before evaluating this multiplication gate. Therefore, we will use the original DN multiplication protocol to evaluate multiplication gates in the first layer.

For a multiplication gate in the second layer, suppose that the two input wires are both the outputs of multiplication gates in the first layer. Let $e_1 - [r_1]_t$ and $e_2 - [r_2]_t$ denote these two output sharings. Now observe that e_1 and e_2 will already be public as part of evaluating the first layer. So to compute a degree-t Shamir sharing of $(e_1 - r_1)(e_2 - r_2)$, all we need is $[r_1 \cdot r_2]_t$. If we can pre-compute and distribute $([r_1]_t, [r_2]_t, [r_1 \cdot r_2]_t)$, we are done! Of course, since r_1 and r_2 are also used in the multiplication gates in the first layer, we simultaneously need to compute degree-$2t$ Shamir sharings of r_1 and r_2 as well. Fortunately, this does not affect the security of the second layer. In other words, the outputs of the first layer feed nicely into the second layer making the second layer non-interactive. At the same time, we are able to ensure that these two different types of multiplication protocols do not destroy the security of each other despite sharing randomness.

As we discussed above, the input sharing of a multiplication gate in the second layer may come from two other places: (1) it may be an input sharing of this two-layer circuit, or (2) it may be an output sharing of an addition gate in the first layer. In both cases, all parties can locally compute this sharing before evaluating the multiplication gates in the first layer. Let $[x]_t$ denote such an input sharing. Note that $[x]_t = 0 - (-[x]_t)$ is already in the Beaver-triple friendly form. Therefore, all the input sharings of multiplication gates in the second layer are in the Beaver-triple friendly form. But now, the problem is that $[x]_t$ is not known before the circuit evaluation starts (unlike $[r_1]_t$ and $[r_2]_t$), and hence $[x]_t$ cannot be part of a Beaver triple pre-computed before the evaluation. Fortunately, as observed earlier, parties hold $[x]_t$ before evaluating any multiplication gates in the first layer. Now our idea is to prepare the Beaver triples for the second layer dependent on $[x]_t$ *in parallel with* the multiplications in the first layer.

After preparing Beaver triples for the second layer and computing the output sharings of the multiplication gates in the first layer, all parties can locally compute the degree-t Shamir sharings associated with the output wires of this two-layer circuit. These sharings will be fed to the next two-layer circuit, which is sufficient to start the evaluation since the original DN multiplication protocol does not require any special property of the input sharings. Therefore in the evaluation of the whole circuit, these two types of multiplication protocols are alternatively used in every

two layers.

Improving the Communication Complexity. While the above helps us make progress, it does not achieve our final goal. In particular, using the original DN protocol requires the communication of 6 elements per party per gate. We note that for multiplications in different layers, we have different requirements:

- For multiplication gates in the first layer, we need the output sharings to have the Beaver-triple friendly form.
- For multiplication gates in the second layer, we compute the Beaver triples in the form of $([a]_t, [b]_t, [c]_t)$. We only need to obtain the degree-t Shamir sharing of $[c]_t$ for each Beaver triple.

Therefore for multiplication gates in the second layer, we can use our improved multiplication protocol to compute Beaver triples, which requires the communication of 4 elements per party per multiplication. For multiplication gates in the first layer, however, $P_{\texttt{king}}$ needs to send the same values to all parties. It seems like our trick of using t-wise independence does not work in this scenario.

Having a closer look at our trick of using t-wise independence, for a multiplication gate handled by an honest party, the secret r of the random double sharings is fixed given the double random sharings used for multiplication gates handled by corrupted parties. Revealing the reconstruction result of $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$ may leak the multiplication result to the adversary. Therefore, to be able to reveal the reconstruction result, r needs to be uniformly random for every multiplication gate. However, we note that r being uniformly random is not equivalent to the pair of double sharings $([r]_t, [r]_{2t})$ being uniformly random.

Therefore, we want to decouple the relation between r and the double sharings. Note that a pair of double sharings $([r]_t, [r]_{2t})$ is equivalent to a pair of sharings $([r]_t, [o]_{2t})$, where the first sharing is a degree-t Shamir sharing of r and the second sharing is a degree-$2t$ Shamir sharing of zero $o = 0$. To see this, given $([r]_t, [r]_{2t})$, we can set $[o]_{2t} := [r]_{2t} - [r]_t$; given $([r]_t, [o]_{2t})$, we can set $[r]_{2t} := [r]_t + [o]_{2t}$. When using a pair of sharings $([r]_t, [o]_{2t})$, the DN multiplication protocol becomes:

1. All parties locally compute $[e]_{2t} := [x]_t \cdot [y]_t + [r]_t + [o]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[e]_{2t}$ and reconstructs the secret e. Then $P_{\texttt{king}}$ sends the value e to all other parties.
3. After receiving e from $P_{\texttt{king}}$, all parties locally compute $[z]_t := e - [r]_t$.

Note that $[o]_{2t}$ is only used to compute $[e]_{2t}$. When $P_{\texttt{king}}$ is an honest party, $[o]_{2t}$ does not need to be a uniformly random degree-$2t$ sharing of 0. Thus, we can use t-wise independent $[o]_{2t}$'s with uniformly random degree-t sharings $[r]_t$'s.

In [28], it has been shown that preparing a random degree-t random sharing requires the communication of 2 elements per party. In Chapter 6.3, following from the same idea of preparing random degree-t Shamir sharings, we show that preparing a random degree-$2t$ sharing of 0 requires the communication of 2 elements per party as well. Then, using our idea of t-wise independence, we expand t random degree-$2t$ sharings of 0 to n sharings with t-wise independence. In this way, the communication cost of preparing correlated-randomness for one multiplication

in the first layer is $2 + 2 \cdot t/n \approx 3$ elements. Including the communication cost of the multiplication protocol in [28], which is 2 elements per party, the overall cost per multiplication in the first layer is 5 elements per party.

Recall that for multiplication gates in the second layer, we will use our improved multiplication protocol to compute Beaver triples, which requires the communication of 4 elements per party per gate. To evaluate the whole circuit, we first partition it into a sequence of two-layer sub-circuits. Then we use the above strategy to evaluate each two-layer sub-circuit in a predetermined topological order. Assuming that the number of multiplication gates in the first layer is roughly the same as the number of multiplication gates in the second layer, the concrete efficiency is $(4 + 5)/2 = 4.5$ elements per party per gate.

Theorem 4.2. *Let n be a positive integer and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n + 1$. For all arithmetic circuit C over $\mathbb{F}$, there is an information-theoretic n-party MPC protocol, which achieves semi-honest security against a semi-honest adversary controlling $t < n/2$ corrupted parties, with communication complexity $O(|C| \cdot n + n^2)$ elements, where $|C|$ is the circuit size. Furthermore, the concrete efficiency is 4.5 elements per party per multiplication gate but halving the number of rounds.*

4.4 Achieving Malicious Security

In [38], Genkin et al. showed that the DN protocol [28] is secure up to an additive attack in the presence of a malicious adversary. An additive attack means that the adversary is able to change the multiplication result by adding an arbitrary fixed value. Security up to an additive attack means that what a malicious adversary can do is to launch additive attacks to multiplication results. As one corollary, the DN protocol provides full privacy of honest parties before reconstructing the output. Therefore, a straightforward strategy to achieve security-with-abort is to (1) run the DN protocol until the output phase, (2) check the correctness of the computation, and (3) reconstruct the output only if the check passes.

In the DN protocol [28], all parties compute a degree-t Shamir sharing for each wire. Since the Shamir secret sharing scheme is linearly homomorphic, addition gates can be evaluated without interaction. Therefore, to achieve security-with-abort, the main task is to verify the multiplications. We first present a verification protocol for multiplication tuples which are under additive attacks. Then we discuss how to compile both of our protocols (the `t-wise` variant and the `round-compression` variant) to achieve security-with-abort by using the above verification protocol. At a high-level,

- For the `t-wise` variant, we will show that the protocol is also secure up to an additive attack. Thus, directly using the verification protocol for multiplication tuples which are under additive attacks is sufficient.
- For the `round-compression` variant, recall that the evaluation is done by first partitioning the circuit into a sequence of two-layer sub-circuits and then evaluating each sub-circuit. We note that the correctness of the computation requires the following two points:
  - All $P_{\texttt{king}}$'s send the same values to all other parties for multiplication gates in the first

layer of all sub-circuits.

- All multiplication tuples are correctly computed.

We will first verify that all parties receive the same values from $P_{\texttt{king}}$'s when evaluating multiplication gates in the first layer of all sub-circuits. Under this condition, we manage to show that the multiplication tuples are secure up to an additive attack. Therefore, we can use the verification for multiplication tuples which are under additive attacks to verify the second point.

Our idea is inspired by two techniques and their natural extensions: the DN multiplication protocol [28] and its extension for an inner-product operation [24], and the batch-wise multiplication verification technique [10] and its extension for inner-product tuples [60]. We first review the batch-wise multiplication verification technique introduced in [10].

4.4.1 Review: Batch-wise Multiplication Verification

This technique is introduced in the work of Ben-Sasson, et al. [10]. It is used to check a batch of multiplication tuples efficiently. Specifically, given m multiplication tuples

$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(m)}]_t, [y^{(m)}]_t, [z^{(m)}]_t),$$

we want to check whether $x^{(i)} \cdot y^{(i)} = z^{(i)}$ for all $i \in \{1, 2, \ldots, m\}$.

The high-level idea is constructing three polynomials $f(\cdot), g(\cdot), h(\cdot)$ such that

$$\forall i \in \{1, 2, \ldots, m\}, f(i) = x^{(i)}, g(i) = y^{(i)}, h(i) = z^{(i)}.$$

Then check whether $f \cdot g = h$. Here $f(\cdot), g(\cdot)$ are degree-$(m-1)$ polynomials so that they can be determined by $\{x^{(i)}\}_{i=1}^{m}, \{y^{(i)}\}_{i=1}^{m}$ respectively. In this case, $h(\cdot)$ should be a degree-$2(m-1)$ polynomial which is determined by $2m-1$ values. To this end, for $i \in \{m+1, \ldots, 2m-1\}$, we need to compute $z^{(i)} = f(i) \cdot g(i)$ so that $h(\cdot)$ can be computed by $\{z^{(i)}\}_{i=1}^{2m-1}$.

All parties first locally compute $[f(\cdot)]_t$ and $[g(\cdot)]_t$ using $\{[x^{(i)}]_t\}_{i=1}^{m}$ and $\{[y^{(i)}]_t\}_{i=1}^{m}$ respectively. Here a degree-t sharing of a polynomial means that each coefficient is secret-shared. For $i \in \{m+1, \ldots, 2m-1\}$, all parties locally compute $[f(i)]_t, [g(i)]_t$ and then compute $[z^{(i)}]_t$ using the multiplication protocol in [28]. Finally, all parties locally compute $[h(\cdot)]_t$ using $\{[z^{(i)}]_t\}_{i=1}^{2m-1}$.

Note that if $x^{(i)} \cdot y^{(i)} = z^{(i)}$ for all $i \in \{1, 2, \ldots, 2m-1\}$, then we have $f \cdot g = h$. Otherwise, we must have $f \cdot g \neq h$. Therefore, it is sufficient to check whether $f \cdot g = h$. Since $h(\cdot)$ is a degree-$2(m-1)$ polynomials, in the case that $f \cdot g = h$, the number of x such that $f(x) \cdot g(x) = h(x)$ holds is at most $2(m-1)$. Thus, it is sufficient to test whether $f(x) \cdot g(x) = h(x)$ for a random x. As a result, this technique compresses m checks of multiplication tuples to a single check of the tuple $([f(x)]_t, [g(x)]_t, [h(x)]_t)$. A secure technique for checking the tuple $([f(x)]_t, [g(x)]_t, [h(x)]_t)$ was given in [10, 60].

The main drawback of this technique is that it requires one additional multiplication operation per tuple. Our idea is to improve this technique so that the check will require fewer multiplication operations.

4.4.2 Extensions

Now we review the two natural extensions of the DN multiplication protocol and the batch-wise multiplication verification technique respectively.

Extension of the DN Multiplication Protocol. In essence, the DN multiplication protocol uses a pair of random double sharings to reduce a degree-$2t$ sharing $[x \cdot y]_{2t}$ to a degree-t sharing $[x \cdot y]_t$. Therefore, an extension of the DN multiplication protocol is used to compute the inner-product of two vectors of the same dimension.

Given two input vectors of sharings $([x_1]_t, \ldots, [x_\ell]_t)$ and $([y_1]_t, \ldots, [y_\ell]_t)$, the goal is to compute a degree-t Shamir sharing $[z]_t$ where

$$z = \sum_{j=1}^{\ell} x_j \cdot y_j.$$

This can be done by using the same strategy as the DN multiplication protocol and in particular, with the *same communication cost*. This is because, just like in the multiplication protocol, here all parties can *locally* compute the shares of the result. These shares are then randomized and sent to $P_{\texttt{king}}$ for degree reduction. More details can be found in Chapter 7.1. This extension is observed in [24].

Extension of the Batch-wise Multiplication Verification. We can use the same strategy as the batch-wise multiplication verification to check the correctness of a batch of *inner-product* tuples.

Specifically, given a set of m inner-product tuples

$$\{(([x_1^{(i)}]_t, \ldots, [x_\ell^{(i)}]_t), ([y_1^{(i)}]_t, \ldots, [y_\ell^{(i)}]_t), [z^{(i)}]_t)\}_{i=1}^{m},$$

we want to check whether $\sum_{j=1}^{\ell} x_j^{(i)} \cdot y_j^{(i)} = z^{(i)}$ for all $i \in \{1, 2, \ldots, m\}$. The only difference is that, for all $j \in \{1, 2, \ldots, \ell\}$, all parties will compute $f_j(\cdot), g_j(\cdot)$ such that

$$\forall i \in \{1, 2, \ldots, m\}, f_j(i) = x_j^{(i)}, g_j(i) = y_j^{(i)},$$

and all parties need to compute $[z^{(i)}]_t = [\sum_{j=1}^{\ell} f_j(i) \cdot g_j(i)]_t$ for all $i \in \{m+1, \ldots, 2m-1\}$, which can be done by the extension of the DN multiplication protocol. Let $h(\cdot)$ be a degree-$2(m-1)$ polynomial such that

$$\forall i \in \{1, 2, \ldots, 2m-1\}, h(i) = z^{(i)}.$$

Then, it is sufficient to test whether $\sum_{j=1}^{\ell} f_j(x) \cdot g_j(x) = h(x)$ for a random x. As a result, this technique compresses m checks of inner-product tuples to a single check of the tuple

$$(([f_1(x)]_t, \ldots, [f_\ell(x)]_t), ([g_1(x)]_t, \ldots, [g_\ell(x)]_t), [h(x)]_t).$$

It is worth noting that the communication cost remains the *same* as the original technique. More details can be found in 7.2. This extension is observed in [60].

Using these Extensions for Reducing the Field Size. We point out that these extensions are not used in any way in the main results of [24, 60]. In [24], the primary purpose of the extension is to check more efficiently in a small field. In more detail, [24] has a "secure MAC" associated with each wire value in the circuit. At a later point, the MACs are verified by computing a linear combination of the value-MAC pairs with random coefficients. Unlike the case in a large field, the random coefficients cannot be made public due to security reasons. Then a computation of a linear combination becomes a computation of an inner-product. [24] relies on the extension of the DN multiplication protocol to efficiently compute the inner-product of two vector of sharings. However we note that with the decrease in the field size, the number of field elements required per gate grows up and hence the concrete efficiency goes down. In [60], the extension of the batch-wise multiplication verification technique is only pointed out as a corollary of independent interest.

4.4.3 Fast Verification for a Batch of Multiplication Tuples

Now we are ready to present our technique. Suppose the multiplication tuples we want to verify are

$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(m)}]_t, [y^{(m)}]_t, [z^{(m)}]_t).$$

The starting idea is to transform these m multiplication tuples into one inner-product tuple. A straightforward way is just setting

$$(([x^{(1)}]_t, [x^{(2)}]_t, \ldots, [x^{(m)}]_t), ([y^{(1)}]_t, [y^{(2)}]_t, \ldots, [y^{(m)}]_t), [z]_t = \sum_{i=1}^{m} [z^{(i)}]_t).$$

However, it is insufficient to check this tuple. For example, if corrupted parties only maliciously behave when computing the first two tuples and cause $z^{(1)}$ to be $x^{(1)} \cdot y^{(1)} + 1$ and $z^{(2)}$ to be $x^{(2)} \cdot y^{(2)} - 1$, we cannot detect it by using this approach. We need to add some randomness so that the resulting tuple will be incorrect with overwhelming probability if any one of the original tuples is incorrect.

Step One: De-Linearization. Our idea is to use two polynomials with coefficients $\{x^{(i)} \cdot y^{(i)}\}$ and $\{z^{(i)}\}$ respectively. Concretely, let

$$\begin{aligned} F(X) &= (x^{(1)} \cdot y^{(1)}) + (x^{(2)} \cdot y^{(2)})X + \ldots + (x^{(m)} \cdot y^{(m)})X^{m-1} \\ G(X) &= z^{(1)} + z^{(2)}X + \ldots + z^{(m)}X^{m-1}. \end{aligned}$$

Then if at least one multiplication tuple is incorrect, we will have $F \neq G$. In this case, the number of x such that $F(x) = G(x)$ is at most $m-1$. Therefore, with overwhelming probability, $F(r) \neq G(r)$ where r is a random element.

All parties will prepare a random degree-t Shamir sharing $[r]_t$ (see Chapter 5.2 for more details) and reconstruct the value r. We set

$$(([x^{(1)}]_t, r[x^{(2)}]_t, \ldots, r^{m-1}[x^{(m)}]_t), ([y^{(1)}]_t, [y^{(2)}]_t, \ldots, [y^{(m)}]_t), [z]_t = \sum_{i=1}^{m} r^{i-1}[z^{(i)}]_t).$$

The above inner-product tuple is what we wish to verify.

Step Two: Dimension-Reduction. Although we only need to verify the correctness of a single inner-product tuple, it is unclear how to do it efficiently. It seems that verifying an inner-product tuple with dimension m would require communicating at least $O(mn)$ field elements. Therefore, instead of directly doing the check, we want to first reduce the dimension of this inner-product tuple.

Towards that end, even though we only have a single inner-product tuple, we will try to take advantage of batch-wise verification of inner-product tuples. Let k be a compression parameter. Our goal is to transform the original tuple of dimension m to be a new tuple of dimension m/k.

To utilize the extension, let $\ell = m/k$ and we separate each of the two input vectors of the original inner-product tuple into k sub-vectors of dimension ℓ. Concretely, For all $i \in \{1, 2, \ldots, k\}$, the i-th sub-vectors from the two input vectors are

$$\begin{aligned}([a_1^{(i)}]_t, \ldots, [a_\ell^{(i)}]_t) &:= (r^{i\cdot\ell-\ell} \cdot [x^{(i\cdot\ell-\ell+1)}]_t, \ldots, r^{i\cdot\ell-1} \cdot [x^{(i\cdot\ell)}]_t) \\ ([b_1^{(i)}]_t, \ldots, [b_\ell^{(i)}]_t) &:= ([y^{(i\cdot\ell-\ell+1)}]_t, \ldots, [y^{(i\cdot\ell)}]_t)\end{aligned}$$

For each $i \in \{1, 2, \ldots, k-1\}$, we compute $[c^{(i)}]_t = [\sum_{j=1}^{\ell} a_j^{(i)} \cdot b_j^{(i)}]_t$ using the extension of the DN multiplication protocol. Then set $[c^{(k)}]_t = [z]_t - \sum_{i=1}^{k-1} [c^{(i)}]_t$. In this way, if the original tuple is incorrect, then at least one of the new inner-product tuples is incorrect.

Finally, we use the extension of the batch-wise multiplication verification technique to compress the check of these k inner-product tuples into one check of a single inner-product tuple. In particular, the resulting tuple has dimension $\ell = m/k$.

Note that the cost of this step is $O(k)$ inner-product operations, which is just $O(k)$ multiplication operations, and a reconstruction of a sharing, which requires $O(n^2)$ elements. After this step, our task is reduced from checking the correctness of an inner-product tuple of dimension m to checking the correctness of an inner-product tuple of dimension ℓ.

Step Three: Recursion and Randomization. We can repeat the second step $\log_k m$ times so that we only need to check the correctness of *a single* multiplication tuple in the end. To simplify the checking process for the last tuple, we make use of additional randomness.

In the last call of the second step, we need to compress the check of k multiplication tuples into one check of a single multiplication tuple. We follow the idea in [60] and include an additional random multiplication tuple as a random mask of these k multiplication tuples. That is, we will compress the check of $k+1$ multiplication tuples in the last call of the second step. In this way, to check the resulting multiplication tuple, all parties can simply reconstruct the sharings and check whether the multiplication is correct. This reconstruction reveals no additional information about the original inner-product tuple because of this added randomness.

The random multiplication tuple is prepared in the following manner.

1. All parties prepare two random sharings $[a]_t, [b]_t$ (see Chapter 5.2 for more details).
2. All parties compute $[c]_t = [a \cdot b]_t$ using the DN multiplication protocol.

Efficiency Analysis. Note that each step of compression requires $O(k)$ inner-product (or multiplication) operations, which requires $O(kn)$ field elements. Also, each step of compression requires to reconstruct a random sharing, which requires $O(n^2)$ field elements. Therefore, the

total amount of communication of verifying m multiplication tuples is $O((kn + n^2) \cdot \log_k m)$ field elements. Since the number of multiplication tuples m is bounded by poly(κ) where κ is the security parameter. If we choose $k = \kappa$, then the cost is just $O(\kappa n + n^2)$ field elements, which is independent of the number of multiplication tuples.

Remark 4.1. *An attractive feature of our approach is that the communication cost is not affected by the field size. To see this, note that the cost of our check only has a sub-linear dependence on the circuit size. Therefore, we can run the check over an extension field of the original field with large enough size, which does not influence the concrete efficiency of our construction.*

As a comparison, the concrete efficiency of both constructions [24, 60] suffer if one uses a small field. This is because in both constructions, the failure probability of the verification depends on the size of the field. For a small field, they need to do the verification several times to acquire the desired security. The same trick does not work because the cost of their checks has a linear dependency on the circuit size.

Relation with the Technique in [13]. We note that our idea is similar to the technique in [13] when it is used to construct MPC protocols. When $n = 3$ and $t = 1$, our construction is very similar to the construction in [13]. For a general n-party setting, the construction in [13] relies on the replicated secret sharings and builds upon the sublinear distributed zero knowledge proofs constructed in [13]. However, the computation cost of the replicated secret sharings goes exponentially in the number of parties. This restricts the construction in [13] to only work for a constant number of parties. On the other hand, we explore the use of the Shamir secret sharing scheme in the n-party setting. Our idea is inspired by the extensions of the DN multiplication protocol [24, 28] and the batch-wise multiplication verification [10, 60]. This allows us to get a positive result without relying on replicated secret sharings. We also note that the construction in [13] requires the sharings (related to the distributed zero knowledge proof) to be robust and verifiable. We simplify this technique by removing the use of a robust secret sharing scheme and a verifiable secret sharing scheme.

Moreover, we explore a recursion trick to further improve the communication complexity of verifying multiplications. Compared with the construction in [13] which requires to communicate $O(\sqrt{|C|})$ bits, we achieve $O((kn + n^2) \cdot \log_k |C| \cdot \kappa)$ bits.

4.4.4 Achieving Malicious Security for Our Two Semi-honest Protocols

Now we show how to use our fast verification protocol to achieve malicious security for both of our two protocols without affecting the concrete efficiency.

For `t-wise` Variant. Recall that in the `t-wise` variant, our idea is to use t-wise independent random double sharings to replace the uniform random double sharings in the DN multiplication protocol. We can show that the `t-wise` variant is also secure up to an additive attack. Thus we can achieve malicious security by the following steps:

1. Run the `t-wise` variant until the output phase.
2. Check the correctness of multiplication tuples using our fast verification protocol.
3. Reconstruct the output only if the check passes.

Compared with the semi-honest protocol, the only additional cost is our fast verification protocol, which has sub-linear communication complexity in the circuit size. Thus, the concrete efficiency of the maliciously secure `t-wise` variant remains 4 field elements per party per multiplication gate.

Theorem 1.1. *Let n be a positive integer and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n+1$. For all arithmetic circuit C over $\mathbb{F}$, there is an information-theoretic n-party MPC protocol, which achieves malicious security (with abort) against a fully malicious adversary controlling $t < n/2$ corrupted parties, with communication complexity $O(|C| \cdot n + n^2 \cdot \kappa + n \cdot \kappa^2)$ elements, where κ is the security parameter and $|C|$ is the circuit size. Furthermore, the concrete efficiency is 4 elements per party per multiplication gate.*

For `round-compression` Variant. Recall that in the `round-compression` variant, the evaluation is done by first partitioning the circuit into a sequence of two-layer sub-circuits and then evaluating each sub-circuit. In particular, the multiplication gates in the first layer requires $P_{\texttt{king}}$ to distribute the *same value* to all other parties. Thus, an adversary can not only launch additive attacks in the multiplication protocols, but also distribute different values to other parties when $P_{\texttt{king}}$ is corrupted. On the other hand, we can still prove that the `round-compression` variant provides full privacy of honest parties before reconstructing the output. Thus, it is sufficient to verify the following two points before reconstructing the output.

- All $P_{\texttt{king}}$'s send the same values to all other parties for multiplication gates in the first layer of all sub-circuits.
- All multiplication tuples are correctly computed.

We will first verify that all parties receive the same values from $P_{\texttt{king}}$'s when evaluating multiplication gates in the first layer of all sub-circuits. This is done by simply computing a random linear combination of all public values and then exchange the result with each other. Note that the communication complexity of the first check is independent of the circuit size.

If the first check passes, we manage to show that the multiplication tuples are secure up to an additive attack. Therefore, we can use our fast verification protocol to check the correctness of all multiplication tuples. In summary, we can achieve malicious security by the following steps:

1. Run the `round-compression` variant until the output phase.
2. Check that all parties receive the same public values from $P_{\texttt{king}}$'s.
3. Check the correctness of multiplication tuples using our fast verification protocol.
4. Reconstruct the output only if the check passes.

Since both checks have sub-linear communication complexity in the circuit size, the concrete efficiency of the maliciously secure `round-compression` variant remains 4.5 field elements per party per multiplication gate.

Theorem 1.2. *Let n be a positive integer and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n+1$. For all arithmetic circuit C over $\mathbb{F}$, there is an information-theoretic n-party MPC protocol, which achieves malicious security (with abort) against a fully malicious adversary controlling $t < n/2$ corrupted parties, with communication complexity $O(|C| \cdot n + n^2 \cdot \kappa + n \cdot \kappa^2)$ elements, where κ is the security parameter and $|C|$ is the circuit size. Furthermore, the concrete efficiency is 4.5 elements per party per multiplication gate but halving the number of rounds (up to a constant*

number of rounds).

Chapter 5

Preliminaries

5.1 Shamir Secret Sharing Scheme

In this work, we use the standard Shamir secret sharing scheme [64]. Recall that n is the number of parties. Let $\alpha_1, \alpha_2, \ldots, \alpha_n$ be n distinct non-zero elements in $\mathbb{F}$.

A *degree-d* Shamir sharing of $x \in \mathbb{F}$ is a vector $(w_1, \ldots, w_n)$ which satisfies that, there exists a polynomial $f(\cdot) \in \mathbb{F}[X]$ of degree at most d such that $f(0) = x$ and $f(\alpha_i) = w_i$ for $i \in \{1, \ldots, n\}$. Each party P_i holds a share w_i and the whole sharing is denoted by $[x]_d$.

Security Properties of the Shamir Secret Sharing Scheme. Recall that t is the number of corrupted parties. The Shamir secret sharing scheme has the following security property:

- A random degree-d Shamir sharing can be reconstructed from any $d+1$ shares, and any d shares are independent of the secret. Therefore, for all $d \geq t$, the shares of a random degree-d Shamir sharing held by corrupted parties are independent of the secret. For all $d < n - t$, the shares of a degree-d Shamir sharing held by honest parties fully determine the secret.

Algebraic Properties of the Shamir Secret Sharing Scheme. In the following, we will utilize two algebraic properties of the Shamir secret sharing scheme. The operations between two sharings are done by letting each party perform the same operation over its own shares.

- Linear Homomorphism: For all $d < n - 1$ and for all degree-d Shamir sharings $[x]_d$ and $[y]_d$,
$$[x + y]_d = [x]_d + [y]_d.$$
- Multiplication: For all $d \leq (n-1)/2$ and for all degree-d Shamir sharings $[x]_d$ and $[y]_d$,
$$[x \cdot y]_{2d} = [x]_d \cdot [y]_d.$$

Terminologies and Remarks. For a degree-k polynomial $f(\cdot) \in \mathbb{G}[X]$, let $c_0, \ldots, c_k$ denote the coefficients of $f(\cdot)$. If all parties hold degree-d sharings of $c_0, \ldots, c_k$, then for all public input $x \in \mathbb{G}$, all parties can locally compute the degree-d sharing $[f(x)]_d$, which is a linear

combination of $[c_0]_d, [c_1]_d, \ldots, [c_k]_d$. Essentially, it means that all parties hold a degree-d sharing of the polynomial $f(\cdot)$. In the following, we use $[f(\cdot)]_d$ to denote a degree-d sharing of the polynomial $f(\cdot)$.

We refer to a pair of sharings $([r]_t, [r]_{2t})$ of the same secret value r as a pair of *double sharings*. Since $n = 2t + 1$ and t parties are corrupted, the rest of $t + 1$ parties are honest. Therefore, the secret value of a degree-t sharing is determined by the shares held by honest parties. Let $\mathcal{H}$ denote the set of honest parties and $\mathcal{C}orr$ denote the set of corrupted parties. Note that once a degree-t sharing is distributed, the secret value is fixed and in particular, corrupted parties can no longer change the secret value even if the sharing is dealt by a corrupted party.

5.2 Generating Random Sharings

We introduce a simple protocol RAND, which comes from [28], to let all parties prepare $t + 1 = O(n)$ random degree-t sharings in the *semi-honest* setting. The functionality is presented in Functionality 5.1.

Figure 5.1: Functionality $\mathcal{F}_{\text{rand}}$

1. $\mathcal{F}_{\text{rand}}$ receives from the adversary the set of shares $\{r_i\}_{i\in\mathcal{C}orr}$.
2. $\mathcal{F}_{\text{rand}}$ randomly samples r. Based on the secret r and the t shares $\{r_i\}_{i\in\mathcal{C}orr}$ of corrupted parties, $\mathcal{F}_{\text{rand}}$ reconstructs the whole sharing $[r]_t$ and distributes the shares of $[r]_t$ to honest parties.

The protocol will utilize a predetermined and fixed Vandermonde matrix of size $n \times (t + 1)$, which is denoted by $\boldsymbol{M}^{\mathrm{T}}$ (therefore $\boldsymbol{M}$ is a $(t + 1) \times n$ matrix). An important property of a Vandermonde matrix is that any $(t+1)\times(t+1)$ submatrix of $\boldsymbol{M}^{\mathrm{T}}$ is *invertible*. The description of RAND appears in Protocol 5.2. The communication complexity of RAND is $O(n^2)$ field elements.

Figure 5.2: Protocol RAND

1. Each party P_i randomly samples a sharing $[s^{(i)}]_t$ and distributes the shares to other parties.
2. All parties locally compute

$$([r^{(1)}]_t, [r^{(2)}]_t, \ldots, [r^{(t+1)}]_t)^{\mathrm{T}} = \boldsymbol{M}([s^{(1)}]_t, [s^{(2)}]_t, \ldots, [s^{(n)}]_t)^{\mathrm{T}}$$

   and output $[r^{(1)}]_t, [r^{(2)}]_t, \ldots, [r^{(t+1)}]_t$.

We show that this protocol securely computes Functionality 5.1 in the presence of *a fully malicious* adversary.

Lemma 5.1. *The protocol* RAND *securely computes the functionality $\mathcal{F}_{rand}$ in the presence of a fully malicious adversary controlling t corrupted parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Recall that $\mathcal{C}orr$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation of RAND. In the first step, when an honest party P_i needs to distribute a random sharing $[s^{(i)}]_t$, $\mathcal{S}$ samples t random elements as the shares of corrupted parties and sends them to the adversary. For each corrupted party P_i, $\mathcal{S}$ receives the shares of $[s^{(i)}]_t$ held by honest parties. Note that $\mathcal{S}$ learns $t+1$ shares of $[s^{(i)}]_t$, which determines the whole sharing. $\mathcal{S}$ computes the shares of $[s^{(i)}]_t$ held by corrupted parties.

In the second step, $\mathcal{S}$ computes the shares of each $[r^{(i)}]_t$ held by corrupted parties and passes these shares to $\mathcal{F}_{\text{rand}}$.

Hybrids Argument. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ changes the way of preparing a random sharing for each honest party P_i:

1. $\mathcal{S}$ first randomly samples the shares of $[s^{(i)}]_t$ held by corrupted parties and sends them to corrupted parties.
2. Then, $\mathcal{S}$ randomly samples the secret $s^{(i)}$. Based on the secret $s^{(i)}$ and the t shares of $[s^{(i)}]_t$ held by corrupted parties, P_i reconstructs the whole sharing $[s^{(i)}]_t$ and distributes the shares to the rest of honest parties.

For each corrupted party P_i, $\mathcal{S}$ locally computes the shares of $[s^{(i)}]_t$ held by corrupted parties.

Note that this does not change the distribution of the random sharings generated by honest parties. The distribution of **Hybrid**$_0$ is identical to the distribution of **Hybrid**$_1$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ omits the second step when preparing a random sharing for each honest party P_i in **Hybrid**$_1$. Recall that in **Hybrid**$_1$, for all $i \in [n]$, $\mathcal{S}$ has computed the shares of $[s^{(i)}]_t$ held by corrupted parties. For each $[r^{(i)}]_t$, $\mathcal{S}$ computes the shares of corrupted parties and sends them to $\mathcal{F}_{\text{rand}}$.

We show that the distribution of **Hybrid**$_2$ is identical to the distribution of **Hybrid**$_1$.

Let $\boldsymbol{M}^{\mathcal{H}}$ denote the sub-matrix of $\boldsymbol{M}$ containing the columns of $\boldsymbol{M}$ with indices in $\mathcal{H}$ and $\boldsymbol{M}^{\mathcal{C}orr}$ denote the sub-matrix of $\boldsymbol{M}$ containing the columns of $\boldsymbol{M}$ with indices in $\mathcal{C}orr$. Let $([s^{(i)}]_t)_{\mathcal{H}}$ denote the vector of the sharings dealt by parties in $\mathcal{H}$ and $([s^{(i)}]_t)_{\mathcal{C}orr}$ denote the vector of the sharings dealt by parties in $\mathcal{C}orr$. Then,

$$\begin{aligned}([r^{(1)}]_t, [r^{(2)}]_t, \ldots, [r^{(t+1)}]_t)^{\mathrm{T}} &= \boldsymbol{M}([s^{(1)}]_t, [s^{(2)}]_t, \ldots, [s^{(n)}]_t)^{\mathrm{T}} \\ &= \boldsymbol{M}^{\mathcal{H}}([s^{(i)}]_t)_{\mathcal{H}}^{\mathrm{T}} + \boldsymbol{M}^{\mathcal{C}orr}([s^{(i)}]_t)_{\mathcal{C}orr}^{\mathrm{T}}.\end{aligned}$$

Note that $\boldsymbol{M}^{\mathcal{H}}$ is a $(t+1) \times (t+1)$ matrix. By the property of Vandermonde matrices, $\boldsymbol{M}^{\mathcal{H}}$ is invertible. Therefore, given the sharings $\{[s^{(i)}]_t\}_{i \in \mathcal{C}orr}$ dealt by corrupted parties, there

is a one-to-one map from $\{[s^{(i)}]_t\}_{i\in\mathcal{H}}$ to $\{[r^{(i)}]_t\}_{i\in[t+1]}$. Note that the only difference between $\mathbf{Hybrid}_1$ and $\mathbf{Hybrid}_2$ is that, in $\mathbf{Hybrid}_1$, $\{[s^{(i)}]_t\}_{i\in\mathcal{H}}$ is randomly generated (based on the shares which have been sent to corrupted parties) while in $\mathbf{Hybrid}_2$, $\mathcal{F}_{\text{rand}}$ directly generates $\{[r^{(i)}]_t\}_{i\in[t+1]}$ based on the shares that corrupted parties should hold. However, this does not change the distribution of the shares of $\{[r^{(i)}]_t\}_{i\in[t+1]}$ held by honest parties. Therefore, the distribution of $\mathbf{Hybrid}_2$ is identical to the distribution of $\mathbf{Hybrid}_1$.

Note that $\mathbf{Hybrid}_2$ is the execution in the ideal world and the distribution of $\mathbf{Hybrid}_2$ is identical to the distribution of $\mathbf{Hybrid}_0$, the execution in the real world. □

5.3 Generating Random Double Sharings

We introduce a simple protocol DOUBLERAND, which comes from [28], to let all parties prepare $t+1 = O(n)$ pairs of random double sharings in the *semi-honest* setting. Recall that a pair of double sharings $([r]_t, [r]_{2t})$ contains two sharings of the same secret value r. Double sharings will be used to evaluate multiplication gates.

The functionality is presented in Functionality 5.3. The description of DOUBLERAND appears in Protocol 5.4. The communication complexity of DOUBLERAND is $O(n^2)$ field elements.

Figure 5.3: Functionality $\mathcal{F}_{\text{doubleRand}}$

1. $\mathcal{F}_{\text{doubleRand}}$ receives from the adversary two sets of shares $\{r_i\}_{i\in\mathcal{C}orr}$ and $\{r'_i\}_{i\in\mathcal{C}orr}$. $\mathcal{F}_{\text{doubleRand}}$ view the first set as the shares of corrupted party for the degree-t sharing, and the second set as the shares for the degree-$2t$ sharing.
2. $\mathcal{F}_{\text{doubleRand}}$ randomly samples r and prepares the double sharings as follows.
   - For the degree-t sharing, based on the secret r and the t shares $\{r_i\}_{i\in\mathcal{C}orr}$ of corrupted parties, $\mathcal{F}_{\text{doubleRand}}$ reconstructs the whole sharing $[r]_t$.
   - For the degree-$2t$ sharing, $\mathcal{F}_{\text{doubleRand}}$ randomly samples t elements as the shares of the first t honest parties. Based on the secret r, the t shares of the first t honest parties, and the t shares $\{r'_i\}_{i\in\mathcal{C}orr}$ of corrupted parties, $\mathcal{F}_{\text{doubleRand}}$ reconstructs the whole sharing $[r]_{2t}$.

   Finally, $\mathcal{F}_{\text{doubleRand}}$ distributes the shares of $([r]_t, [r]_{2t})$ to honest parties.

We show that this protocol securely computes Functionality 5.3 in the presence of *a fully malicious* adversary.

Lemma 5.2. *The protocol* DOUBLERAND *securely computes the functionality $\mathcal{F}_{\text{doubleRand}}$ in the presence of a fully malicious adversary controlling t corrupted parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Recall that $\mathcal{C}orr$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Figure 5.4: Protocol DOUBLERAND

1. Each party P_i randomly samples a pair of double sharings $([s^{(i)}]_t, [s^{(i)}]_{2t})$ and distributes the shares to other parties.
2. All parties locally compute

$$\begin{aligned}([r^{(1)}]_t, [r^{(2)}]_t, \ldots, [r^{(t+1)}]_t)^{\mathrm{T}} &= \boldsymbol{M}([s^{(1)}]_t, [s^{(2)}]_t, \ldots, [s^{(n)}]_t)^{\mathrm{T}} \\ ([r^{(1)}]_{2t}, [r^{(2)}]_{2t}, \ldots, [r^{(t+1)}]_{2t})^{\mathrm{T}} &= \boldsymbol{M}([s^{(1)}]_{2t}, [s^{(2)}]_{2t}, \ldots, [s^{(n)}]_{2t})^{\mathrm{T}}\end{aligned}$$

   and output $([r^{(1)}]_t, [r^{(1)}]_{2t}), ([r^{(2)}]_t, [r^{(2)}]_{2t}), \ldots, ([r^{(t+1)}]_t, [r^{(t+1)}]_{2t})$.

Simulation of DOUBLERAND. In the first step, when an honest party P_i needs to distribute a pair of random double sharings $([s^{(i)}]_t, [s^{(i)}]_{2t})$, for each corrupted party P_j, $\mathcal{S}$ samples 2 random elements as its shares of $([s^{(i)}]_t, [s^{(i)}]_{2t})$ and sends them to the adversary. For each corrupted party P_i, $\mathcal{S}$ receives the shares of $([s^{(i)}]_t, [s^{(i)}]_{2t})$ held by honest parties. Note that $\mathcal{S}$ learns $t+1$ shares of $[s^{(i)}]_t$, which determines the whole sharing. $\mathcal{S}$ computes the secret $s^{(i)}$ and the shares of $[s^{(i)}]_t$ held by corrupted parties. For $[s^{(i)}]_{2t}$, $\mathcal{S}$ samples a random degree-$2t$ sharing based on the secret $s^{(i)}$ and the shares held by honest parties, and views this degree-$2t$ sharing as the one distributed by P_i.

In the second step, $\mathcal{S}$ computes the shares of each pair $([r^{(i)}]_t, [r^{(i)}]_{2t})$ held by corrupted parties and passes these shares to $\mathcal{F}_{\text{doubleRand}}$.

Hybrids Argument. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ changes the way of preparing a pair of random sharings for each honest party P_i:

1. $\mathcal{S}$ first randomly samples the shares of $([s^{(i)}]_t, [s^{(i)}]_{2t})$ held by corrupted parties and sends them to corrupted parties.
2. Then, $\mathcal{S}$ randomly samples the secret $s^{(i)}$. Based on the secret $s^{(i)}$ and the t shares of $[s^{(i)}]_t$ held by corrupted parties, P_i reconstructs the whole sharing $[s^{(i)}]_t$. Based on the secret $s^{(i)}$ and the shares of $[s^{(i)}]_{2t}$ held by corrupted parties, P_i samples a random degree-$2t$ sharing $[s^{(i)}]_{2t}$ (in the same way as $\mathcal{F}_{\text{doubleRand}}$ in Step 2). Then $\mathcal{S}$ distributes the shares to the rest of honest parties.

For each corrupted party P_i, $\mathcal{S}$ locally computes the shares of $([s^{(i)}]_t, [s^{(i)}]_{2t})$ held by corrupted parties.

Note that this does not change the distribution of the random double sharings generated by honest parties. The distribution of **Hybrid**$_0$ is identical to the distribution of **Hybrid**$_1$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ omits the second step when preparing a pair of random sharings for each honest party P_i in **Hybrid**$_1$. Recall that in **Hybrid**$_1$, for all $i \in [n]$, $\mathcal{S}$ has computed the

shares of $([s^{(i)}]_t, [s^{(i)}]_{2t})$ held by corrupted parties. For each pair $([r^{(i)}]_t, [r^{(i)}]_{2t})$, $\mathcal{S}$ computes the shares of corrupted parties and sends them to $\mathcal{F}_{\text{doubleRand}}$.

We show that the distribution of $\mathbf{Hybrid}_2$ is identical to the distribution of $\mathbf{Hybrid}_1$.

Let $\boldsymbol{M}^{\mathcal{H}}$ denote the sub-matrix of $\boldsymbol{M}$ containing the columns of $\boldsymbol{M}$ with indices in $\mathcal{H}$ and $\boldsymbol{M}^{\mathcal{C}orr}$ denote the sub-matrix of $\boldsymbol{M}$ containing the columns of $\boldsymbol{M}$ with indices in $\mathcal{C}orr$. Let $([s^{(i)}]_t)_{\mathcal{H}}, ([s^{(i)}]_{2t})_{\mathcal{H}}$ denote the vectors of the sharings dealt by parties in $\mathcal{H}$ and $([s^{(i)}]_t)_{\mathcal{C}orr}$, $([s^{(i)}]_{2t})_{\mathcal{C}orr}$ denote the vectors of the sharings dealt by parties in $\mathcal{C}orr$. Then,

$$\begin{aligned}
([r^{(1)}]_t, [r^{(2)}]_t, \ldots, [r^{(t+1)}]_t)^{\mathrm{T}} &= \boldsymbol{M}([s^{(1)}]_t, [s^{(2)}]_t, \ldots, [s^{(n)}]_t)^{\mathrm{T}} \\
&= \boldsymbol{M}^{\mathcal{H}}([s^{(i)}]_t)^{\mathrm{T}}_{\mathcal{H}} + \boldsymbol{M}^{\mathcal{C}orr}([s^{(i)}]_t)^{\mathrm{T}}_{\mathcal{C}orr} \\
([r^{(1)}]_{2t}, [r^{(2)}]_{2t}, \ldots, [r^{(t+1)}]_{2t})^{\mathrm{T}} &= \boldsymbol{M}([s^{(1)}]_{2t}, [s^{(2)}]_{2t}, \ldots, [s^{(n)}]_{2t})^{\mathrm{T}} \\
&= \boldsymbol{M}^{\mathcal{H}}([s^{(i)}]_{2t})^{\mathrm{T}}_{\mathcal{H}} + \boldsymbol{M}^{\mathcal{C}orr}([s^{(i)}]_{2t})^{\mathrm{T}}_{\mathcal{C}orr}
\end{aligned}$$

Note that $\boldsymbol{M}^{\mathcal{H}}$ is a $(t+1) \times (t+1)$ matrix. By the property of Vandermonde matrices, $\boldsymbol{M}^{\mathcal{H}}$ is invertible. Therefore, given the sharings $\{([s^{(i)}]_t, [s^{(i)}]_{2t})\}_{i\in\mathcal{C}orr}$ dealt by corrupted parties, there is a one-to-one map from $\{([s^{(i)}]_t, [s^{(i)}]_{2t})\}_{i\in\mathcal{H}}$ to $\{([r^{(i)}]_t, [r^{(i)}]_{2t})\}_{i\in[t+1]}$. Note that the only difference between $\mathbf{Hybrid}_1$ and $\mathbf{Hybrid}_2$ is that, in $\mathbf{Hybrid}_1$, $\{([s^{(i)}]_t, [s^{(i)}]_{2t})\}_{i\in\mathcal{H}}$ is randomly generated (based on the shares which have been sent to corrupted parties) while in $\mathbf{Hybrid}_2$, $\mathcal{F}_{\text{doubleRand}}$ directly generates $\{([r^{(i)}]_t, [r^{(i)}]_{2t})\}_{i\in[t+1]}$ based on the shares that corrupted parties should hold. However, this does not change the distribution of the shares of $\{[r^{(i)}]_t\}_{i\in[t+1]}$ held by honest parties. To see this, note that for any double sharings $\{([r^{(i)}]_t, [r^{(i)}]_{2t})\}_{i\in[t+1]}$ generated by $\mathcal{F}_{\text{doubleRand}}$, we can compute back to a set of valid double sharings $\{([s^{(i)}]_t, [s^{(i)}]_{2t})\}_{i\in\mathcal{H}}$. Therefore, the distribution of $\mathbf{Hybrid}_2$ is identical to the distribution of $\mathbf{Hybrid}_1$.

Note that $\mathbf{Hybrid}_2$ is the execution in the ideal world and the distribution of $\mathbf{Hybrid}_2$ is identical to the distribution of $\mathbf{Hybrid}_0$, the execution in the real world. □

5.4 Generating Random Coins

Relying on $\mathcal{F}_{\text{rand}}$, we show how to securely generate a random field element. The functionality $\mathcal{F}_{\text{coin}}$ is presented in Functionality 5.5. The description of COIN appears in Protocol 5.6. The communication complexity of COIN is $O(n^2)$ field elements.

Figure 5.5: Functionality $\mathcal{F}_{\text{coin}}$

1. $\mathcal{F}_{\text{coin}}$ samples a random field element r.
2. $\mathcal{F}_{\text{coin}}$ sends r to the adversary.
   - If the adversary replies `continue`, $\mathcal{F}_{\text{coin}}$ sends r to honest parties.
   - If the adversary replies `abort`, $\mathcal{F}_{\text{coin}}$ sends `abort` to honest parties.

Lemma 5.3. *The protocol* COIN *securely computes $\mathcal{F}_{coin}$ with abort in the $\mathcal{F}_{rand}$-hybrid model in the presence of a fully malicious adversary controlling t corrupted parties.*

Figure 5.6: Protocol COIN

1. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare a random sharing $[r]_t$.
2. Every party P_i sends its share of $[r]_t$ to all other parties. After receiving all the shares, P_i checks that whether $[r]_t$ is a valid sharing, i.e., all the shares lie on a polynomial of degree at most t.
   - If true, P_i reconstructs the secret r and takes it as output.
   - Otherwise, P_i sends `abort` to all other parties and aborts.

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Recall that $\mathcal{C}orr$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation of COIN. In the beginning, $\mathcal{S}$ receives the random element r from $\mathcal{F}_{\text{coin}}$. When invoking $\mathcal{F}_{\text{rand}}$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ and receives a set of shares $\{r_i\}_{i\in\mathcal{C}orr}$ from the adversary. Based on the secret r, and the shares $\{r_i\}_{i\in\mathcal{C}orr}$ held by corrupted parties, $\mathcal{S}$ reconstructs the whole sharing $[r]_t$.

After obtaining the whole sharing $[r]_t$, $\mathcal{S}$ can faithfully follow the protocol in COIN. If an honest party aborts, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{coin}}$. Otherwise, $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\text{coin}}$.

Analysis of the security. Note that the only difference between the real world execution and the ideal world execution is that, in the real world, the secret value r is randomly sampled by $\mathcal{F}_{\text{rand}}$, while in the ideal world, r is received from $\mathcal{F}_{\text{coin}}$. However, in both $\mathcal{F}_{\text{rand}}$ and $\mathcal{F}_{\text{coin}}$, r is randomly sampled. Therefore, the distributions of both executions are identical. □

Chapter 6

Efficient Semi-Honest MPC Protocols

In this chapter, we will introduce two improvements to the semi-honest DN protocol in [28].

- The first improvement reduces the communication cost per multiplication gate per party from 6 elements to 4 elements.
- The second improvement reduces the communication cost per multiplication gate per party from 6 elements to 4.5 elements *and reduce the number of rounds by a factor of* 2.

Our core idea is to reuse the correlated-randomness prepared for multiplication gates.

We first give a short review of the construction in [28]. Then we introduce our two improvements.

6.1 Review of the Semi-Honest DN Protocol in [28]

At a high-level, the semi-honest protocol in [28] computes a degree-t Shamir sharing for each wire. Since the Shamir secret sharing scheme is linear homomorphic, addition gates can be evaluated without interaction. Therefore, the main concern is the evaluation of multiplication gates.

The Multiplication Protocol in [28]. For a multiplication gate, given two input degree-t Shamir sharings $[x]_t, [y]_t$, the goal is to compute a degree-t Shamir sharing of the multiplication result $[x \cdot y]_t$. We model the ideal functionality $\mathcal{F}_{\text{mult}}$ in Functionality 6.1.

In [28], to evaluate a multiplication gate, all parties need to prepare a pair of random double sharings $([r]_t, [r]_{2t})$. This is done by invoking $\mathcal{F}_{\text{doubleRand}}$ introduced in Chapter 5.2. Recall that the amortized communication complexity of the instanciation of $\mathcal{F}_{\text{doubleRand}}$ in [28] is 4 elements per party.

For a multiplication gate, suppose the input sharings are denoted by $[x]_t, [y]_t$. To compute $[z]_t := [x \cdot y]_t$, a pair of random double sharings $([r]_t, [r]_{2t})$ is consumed. All parties first agree on a special party $P_{\texttt{king}}$. $P_{\texttt{king}}$ will help do the reconstruction in the multiplication protocol. Then, all parties run the following steps:

1. All parties locally compute $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$.

Figure 6.1: Functionality $\mathcal{F}_{\text{mult}}$

1. Let $[x]_t, [y]_t$ denote the input sharings. $\mathcal{F}_{\text{mult}}$ receives from honest parties their shares of $[x]_t, [y]_t$. Then $\mathcal{F}_{\text{mult}}$ reconstructs the secrets x, y. $\mathcal{F}_{\text{mult}}$ further computes the shares of $[x]_t, [y]_t$ held by corrupted parties, and sends these shares to the adversary.
2. $\mathcal{F}_{\text{mult}}$ receives from the adversary a set of shares $\{z_i\}_{i \in \mathcal{C}orr}$.
3. $\mathcal{F}_{\text{mult}}$ computes $x \cdot y$. Based on the secret $z := x \cdot y$ and the t shares $\{z_i\}_{i \in \mathcal{C}orr}$, $\mathcal{F}_{\text{mult}}$ reconstructs the whole sharing $[z]_t$ and distributes the shares of $[z]_t$ to honest parties.

2. $P_{\texttt{king}}$ collects all shares of $[e]_{2t}$ and reconstructs the secret e. Then $P_{\texttt{king}}$ sends the value e to all other parties.
3. After receiving e from $P_{\texttt{king}}$, all parties locally compute $[z]_t := e - [r]_t$.

The correctness follows from the properties of the Shamir secret sharing scheme. Note that each party needs to send an element to $P_{\texttt{king}}$, and $P_{\texttt{king}}$ needs to send an element to each party. The communication complexity of this protocol is 2 elements per party. Including the communication cost for preparing double sharings, the overall cost per multiplication gate is 6 elements per party. We formally describe the DN multiplication protocol in Protocol 6.2.

Figure 6.2: Protocol DN-MULT

1. Suppose $[x]_t, [y]_t$ are the input degree-t Shamir sharings of the multiplication gate.
2. All parties invoke $\mathcal{F}_{\text{doubleRand}}$ to prepare a pair of random double sharings $([r]_t, [r]_{2t})$.
3. All parties locally compute $[e]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$.
4. $P_{\texttt{king}}$ collects all shares and reconstructs the secret $e = x \cdot y + r$. Then $P_{\texttt{king}}$ sends e to all other parties.
5. All parties locally compute $[z]_t = e - [r]_t$.

The Main Protocol in [28]. Based on $\mathcal{F}_{\text{mult}}$, Damgård and Nielsen construct the DN protocol as follows (described in Protocol 6.3).

We have the following theorem from [28].

Theorem 6.1 ([28])**.** *Let n be a positive integer and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n + 1$. For all arithmetic circuit C over $\mathbb{F}$, there is an information-theoretic n-party MPC protocol, which achieves semi-honest security against a semi-honest adversary controlling $t < n/2$ corrupted parties, with communication complexity $O(|C| \cdot n + n^2)$ elements, where $|C|$ is the circuit size. Furthermore, the concrete efficiency is 6 elements per party per multiplication gate.*

Figure 6.3: Protocol DN-MAIN

1. **Input Phase**. Let $\mathsf{Client}_1, \ldots, \mathsf{Client}_c$ denote the clients who provide inputs. For every input gate of Client_i, Client_i samples a random degree-t Shamir sharing of its input, $[x]_t$, and distributes the shares to all parties.
2. **Evaluation Phase**. All parties evaluate the circuit layer by layer.
   - For each addition gate with input sharings $[x]_t, [y]_t$, all parties locally compute the output sharing $[z]_t := [x]_t + [y]_t$.
   - For each multiplication gate with input sharings $[x]_t, [y]_t$, all parties invoke $\mathcal{F}_{\text{mult}}$ to compute the output sharing $[z]_t = [x \cdot y]_t$.
3. **Output Phase**. For each output gate of Client_i, let $[x]_t$ denote the input sharing. All parties send their shares of $[x]_t$ to Client_i to let Client_i reconstruct x.

6.2 Reducing the Communication Complexity via t-wise Independence

Our first improvement comes from a new protocol for $\mathcal{F}_{\text{mult}}$. The amortized communication cost of our new protocol is 4 elements per party. Our new protocol is based on the multiplication protocol in [28].

In [44], Goyal et al. observe that in the second step, $P_{\texttt{king}}$ can alternatively distribute a degree-t Shamir sharing $[e]_t$. Then in the last step, all parties can still compute $[z]_t := [e]_t - [r]_t$. Furthermore, since e does not need to be private, $P_{\texttt{king}}$ can set the shares of (a predetermined set of) t parties to be 0 in $[e]_t$. This means that $P_{\texttt{king}}$ need not to communication these shares at all, reducing the communication by half. This observation allows Goyal et al. to reduce the communication cost from 6 elements to 5.5 elements.

6.2.1 Our Observation

We observe that in the fourth step of Protocol DN-MULT, $P_{\texttt{king}}$ can alternatively distribute a degree-t Shamir sharing $[e]_t$. Then in the last step, all parties can still compute $[z]_t := [e]_t - [r]_t$. By requiring $P_{\texttt{king}}$ to generate a random sharing $[e]_t$, when $P_{\texttt{king}}$ is an honest party, corrupted parties only receive t shares of a random degree-t sharing $[e]_t$ from $P_{\texttt{king}}$, which are uniform and independent of the secret. As discussed in Chapter 4.2, it means that we do not need to use uniform double sharings when $P_{\texttt{king}}$ is honest.

For n multiplication gates, our idea is to let each party behave as $P_{\texttt{king}}$ for one multiplication gate. Note that only t out of n multiplications are handled by corrupted $P_{\texttt{king}}$'s. To make sure that all parties still use a pair of random double sharings when $P_{\texttt{king}}$ is corrupted, the n pairs of double sharings for these n multiplication gates only need to be t-wise independent. To this end, we will first generate t pairs of random double sharings, and then expand them to n pairs of double sharings with t-wise independence.

Specifically, all parties agree on an $n \times t$ Vandermonde matrix $\boldsymbol{M}$. Let

$$([r^{(1)}]_t, [r^{(1)}]_{2t}), \ldots, ([r^{(t)}]_t, [r^{(t)}]_{2t})$$

be t pairs of random double sharings prepared by $\mathcal{F}_{\text{doubleRand}}$. All parties execute EXPAND (Protocol 6.4) to expand these t pairs into n pairs of t-wise independent double sharings.

Figure 6.4: Protocol EXPAND

1. All parties agree on an $n \times t$ Vandermonde matrix $\boldsymbol{M}$. All parties locally compute

$$\begin{aligned} ([\tilde{r}^{(1)}]_t, \ldots, [\tilde{r}^{(n)}]_t)^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_t, \ldots, [r^{(t)}]_t)^{\mathrm{T}} \\ ([\tilde{r}^{(1)}]_{2t}, \ldots, [\tilde{r}^{(n)}]_{2t})^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_{2t}, \ldots, [r^{(t)}]_{2t})^{\mathrm{T}} \end{aligned}$$

2. All parties output $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t}, P_i)\}_{i=1}^{n}$, where $([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t}, P_i)$ will be used for a multiplication gate handled by P_i.

Recall that $\mathcal{C}orr$ denotes the set of all corrupted parties. By the property of Vandermonde matrices, there is a one-to-one map from $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})\}_{i \in \mathcal{C}orr}$ to $\{[r^{(i)}]_t, [r^{(i)}]_{2t}\}_{i=1}^{t}$. Thus, $\{([\tilde{r}^{(i)}]_t, [\tilde{r}^{(i)}]_{2t})\}_{i \in \mathcal{C}orr}$ are t pairs of random double sharings.

6.2.2 ATLAS Multiplication Protocol

To evaluate a multiplication gate, a pair of double sharings $([r]_t, [r]_{2t}, P_i)$ is consumed. All parties execute MULT (Protocol 6.5).

Figure 6.5: Protocol MULT

1. Let $([r]_t, [r]_{2t}, P_i)$ be the random double sharings which will be used in the protocol. Let $[x]_t, [y]_t$ denote the input sharings.
2. All parties locally compute $[e]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$.
3. P_i collects all shares and reconstructs the secret $e = x \cdot y + r$. Then P_i randomly generates a degree-t Shamir sharing $[e]_t$ and distributes the shares to other parties.
4. All parties locally compute $[z]_t = [e]_t - [r]_t$.

To show the security of ATLAS multiplication protocol, we consider the scenario where all parties evaluate *a sequence of* N multiplication gates. In particular, the input sharings of each multiplication gate can depend on the input sharings or output sharings of the previous multiplication gates. The functionality $\mathcal{F}'_{\text{mult}}$ appears in Functionality 6.6, which invokes $\mathcal{F}_{\text{mult}}$ for each multiplication gate. One can view $\mathcal{F}'_{\text{mult}}$ as an interface of $\mathcal{F}_{\text{mult}}$. It allows us to replace the invocation of $\mathcal{F}_{\text{mult}}$ in the semi-honest DN protocol [28] by the invocation of $\mathcal{F}'_{\text{mult}}$, and thus

directly use ATLAS multiplication protocol in the semi-honest DN protocol [44]. The protocol ATLAS-MULT appears in Protocol 6.7.

Figure 6.6: Functionality $\mathcal{F}'_{\text{mult}}$

1. $\mathcal{F}'_{\text{mult}}$ receives N from all parties.
2. From $i = 1$ to N, let $[x^{(i)}]_t, [y^{(i)}]_t$ denote the input sharings of the i-th multiplication gate. $\mathcal{F}'_{\text{mult}}$ invokes $\mathcal{F}_{\text{mult}}$ on $[x^{(i)}]_t, [y^{(i)}]_t$.

Figure 6.7: Protocol ATLAS-MULT

1. All parties set N to be the number of multiplication gates to be evaluated.
2. All parties invoke $\mathcal{F}_{\text{doubleRand}}$ to prepare $N \cdot t/n$ pairs of random double sharings, and invoke EXPAND to obtain N pairs of double sharings in the form of $([r]_t, [r]_{2t}, P_j)$
3. From $i = 1$ to N, let $[x^{(i)}]_t, [y^{(i)}]_t$ denote the input sharings of the i-th multiplication gate. Suppose $([r]_t, [r]_{2t}, P_j)$ is the first pair of unused double sharings. All parties invoke MULT on $[x^{(i)}]_t, [y^{(i)}]_t$ and $([r]_t, [r]_{2t}, P_j)$.

Lemma 6.1. *The protocol* ATLAS-MULT *securely computes $\mathcal{F}'_{mult}$ in the $\mathcal{F}_{doubleRand}$-hybrid model in the presence of a semi-honest adversary controlling t corrupted parties.*

Proof. Recall that $\mathcal{C}orr$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties. We first show that the random degree-$2t$ sharing $[r]_{2t}$ output by $\mathcal{F}_{\text{doubleRand}}$ satisfies that the shares of honest parties are uniformly random, and are independent of the shares chosen by the adversary. Recall that $\mathcal{F}_{\text{doubleRand}}$ receives from the adversary two sets of shares $\{r_i\}_{i\in\mathcal{C}orr}, \{r'_i\}_{i\in\mathcal{C}orr}$ and randomly samples $([r]_t, [r]_{2t})$ such that the shares of $[r]_t, [r]_{2t}$ held by corrupted parties are $\{r_i\}_{i\in\mathcal{C}orr}, \{r'_i\}_{i\in\mathcal{C}orr}$ respectively. Consider the following sampling process:

1. $\mathcal{F}_{\text{doubleRand}}$ randomly samples $t+1$ shares as the shares of $[r]_{2t}$ held by honest parties. Then, $\mathcal{F}_{\text{doubleRand}}$ reconstructs the whole sharing $[r]_{2t}$ using the shares of honest parties and the shares $\{r'_i\}_{\mathcal{C}orr}$ of corrupted parties, and computes the secret r.
2. $\mathcal{F}_{\text{doubleRand}}$ reconstructs the whole sharing $[r]_t$ based on the secret r and the shares $\{r_i\}_{i\in\mathcal{C}orr}$ of corrupted parties.

Note that the above process output a pair of random double sharings with the same distribution as that described in the original $\mathcal{F}_{\text{doubleRand}}$. However, the shares of $[r]_{2t}$ held by honest parties are randomly chosen in the first step and are independent of the shares $\{r_i\}_{i\in\mathcal{C}orr}, \{r'_i\}_{i\in\mathcal{C}orr}$.

Recall that in EXPAND, all parties compute

$$\begin{aligned}([\tilde{r}^{(1)}]_t, \ldots, [\tilde{r}^{(n)}]_t)^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_t, \ldots, [r^{(t)}]_t)^{\mathrm{T}} \\ ([\tilde{r}^{(1)}]_{2t}, \ldots, [\tilde{r}^{(n)}]_{2t})^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_{2t}, \ldots, [r^{(t)}]_{2t})^{\mathrm{T}},\end{aligned}$$

where $\boldsymbol{M}$ is a Vandermonde matrix. From the above argument, the shares of $\{[r^{(i)}]_{2t}\}_{i=1}^{t}$ held by honest parties are uniformly random. According to the property of Vandermonde matrices, there is a one-to-one map from $\{[r^{(i)}]_{2t}\}_{i=1}^{t}$ to $\{[\tilde{r}^{(i)}]_{2t}\}_{i\in\mathcal{C}orr}$. Thus, the shares of $\{[\tilde{r}^{(i)}]_{2t}\}_{i\in\mathcal{C}orr}$ held by honest parties are uniformly random. This means that for all double sharings in the form of $([r]_t, [r]_{2t}, P_j)$ output by EXPAND, where P_j is a corrupted party, the shares of $[r]_{2t}$ held by honest parties are uniformly random.

Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties.

Simulation for ATLAS-MULT. In the first step, $\mathcal{S}$ emulates $\mathcal{F}_{\text{doubleRand}}$ and receives the shares of random double sharings held by corrupted parties. Then $\mathcal{S}$ follows EXPAND and computes the shares of the double sharings in the form of $([r]_t, [r]_{2t}, P_j)$ held by corrupted parties.

In the second step, we describe the strategy of $\mathcal{S}$ for each invocation of MULT.

1. Let $[x^{(i)}]_t, [y^{(i)}]_t$ denote the input sharings. $\mathcal{S}$ receives from $\mathcal{F}'_{\text{mult}}$ the shares of $[x^{(i)}]_t, [y^{(i)}]_t$ held by corrupted parties. Let $([r]_t, [r]_{2t}, P_j)$ be the first pair of unused double sharings. Recall that $\mathcal{S}$ has learnt the shares of $[r]_t, [r]_{2t}$ held by corrupted parties.
2. $\mathcal{S}$ computes the shares of $[e^{(i)}]_{2t} := [x^{(i)}]_t \cdot [y^{(i)}]_t + [r]_{2t}$ held by corrupted parties. If P_j is corrupted, $\mathcal{S}$ samples random elements as shares of $[e^{(i)}]_{2t}$ of honest parties.
3. Depending on whether P_j is honest, there are two cases:
   - If P_j is honest, $\mathcal{S}$ receives the shares from corrupted parties. For $[e^{(i)}]_t$, $\mathcal{S}$ samples random elements as the shares of corrupted parties.
   - If P_j is corrupted, $\mathcal{S}$ sends the shares of $[e^{(i)}]_{2t}$ of honest parties to P_j. Then $\mathcal{S}$ receives the shares of $[e^{(i)}]_t$ of honest parties from P_j and computes the shares held by corrupted parties.
4. In the last step, $\mathcal{S}$ computes the shares of $[z^{(i)}]_t := [e^{(i)}]_t - [\tilde{r}^{(i)}]_t$ held by corrupted parties. Then $\mathcal{S}$ sends the shares of $[z^{(i)}]_t$ held by corrupted parties to $\mathcal{F}'_{\text{mult}}$.

Analysis of the Security. Note that there are two differences between the real world execution and the ideal world execution:

- When a multiplication gate is handled by an honest party P_j, $\mathcal{S}$ samples random field elements as the shares of $[e]_t$ of corrupted parties. Note that in the real world, an honest party P_j will generate a random degree-t Shamir sharing $[e]_t$, which satisfies that the shares of corrupted parties are uniformly random. Thus, the distribution of the shares of $[e]_t$ held by corrupted parties is identical in both worlds.
- When a multiplication gate is handled by a corrupted party P_j, $\mathcal{S}$ samples random field elements as the shares of $[e]_{2t}$ of honest parties. Recall that $[e]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$. As we argued above, when P_j is corrupted, the shares of $[r]_{2t}$ held by honest parties are uniformly random. Therefore, the shares of $[e]_2 t$ held by honest parties are also uniformly random. Thus, the distribution of the shares of $[e]_{2t}$ held by honest parties is identical in both worlds.

Finally, regarding the outputs of honest parties, note that they are determined by the secrets and the shares of corrupted parties. Since $\mathcal{S}$ computes the shares of corrupted parties and sends them

to $\mathcal{F}'_{\text{mult}}$, the distribution of the outputs of honest parties is identical in both worlds. We conclude that the distribution of both worlds are identical. □

6.2.3 Using $\mathcal{F}'_{\text{mult}}$ in the Semi-Honest DN protocol in [28]

In the semi-honest DN protocol in [28], all parties invoke $\mathcal{F}_{\text{mult}}$ for each multiplication gate. Note that $\mathcal{F}'_{\text{mult}}$ invoke $\mathcal{F}_{\text{mult}}$ for each multiplication. Therefore, we view $\mathcal{F}'_{\text{mult}}$ as an interface of $\mathcal{F}_{\text{mult}}$. All parties initialize $\mathcal{F}'_{\text{mult}}$ in the beginning of the protocol with the number of multiplications they need to compute (which is determined by the circuit). Then we replace each invocation of $\mathcal{F}_{\text{mult}}$ by $\mathcal{F}'_{\text{mult}}$.

Note that every t pairs of random double sharings generated by $\mathcal{F}_{\text{doubleRand}}$ are expanded to n pairs of double sharings. Therefore, the communication cost per pair of double sharings is $4 \cdot t/n \approx 2$ elements per party. The overall cost per multiplication gate is 4 elements per party. Therefore, when using ATLAS-MULT to instantiate $\mathcal{F}'_{\text{mult}}$, we obtain a semi-honest MPC protocol with communication complexity $O(|C| \cdot n + n^2)$ field elements. In particular, the concrete efficiency per multiplication gate is 4 elements per party.

Theorem 4.1. *Let n be a positive integer and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n + 1$. For all arithmetic circuit C over $\mathbb{F}$, there is an information-theoretic n-party MPC protocol, which achieves semi-honest security against a semi-honest adversary controlling $t < n/2$ corrupted parties, with communication complexity $O(|C| \cdot n + n^2)$ elements, where $|C|$ is the circuit size. Furthermore, the concrete efficiency is 4 elements per party per multiplication gate.*

6.3 Reducing the Number of Rounds via Beaver Triples

For the semi-honest DN protocol in [28], multiplication gates in the same layer of the circuit are evaluated in parallel. Therefore, the number of rounds is linear in the depth of the circuit. To further improve the concrete efficiency, we pay our attention to the round complexity. In this part, we show that multiplication gates in a two-layer circuit can be evaluated in parallel. It allows us to reduce the number of rounds by a factor of 2. The amortized communication cost per multiplication gate is 4.5 elements per party.

6.3.1 An Overview of Our Approach

We first start with a two-layer circuit. At a high-level, we use Beaver triples to evaluate multiplications in the second layer. Recall that a Beaver triple consists of three degree-t Shamir sharings $([a]_t, [b]_t, [c]_t)$ such that $c = a \cdot b$. Usually, a Beaver triple is used to transform one multiplication to two reconstructions. Concretely, given two sharings $[x]_t, [y]_t$, suppose we want to compute $[z]_t$ such that $z = x \cdot y$. Since

$$\begin{aligned} z &= x \cdot y = (x + a - a) \cdot (y + b - b) \\ &= (x + a) \cdot (y + b) - (x + a) \cdot b - (y + b) \cdot a + a \cdot b, \end{aligned}$$

we can compute

$$[z]_t := (x + a) \cdot (y + b) - (x + a) \cdot [b]_t - (y + b) \cdot [a]_t + [c]_t.$$

Therefore, the task of computing $[z]_t$ becomes to reconstruct two degree-t Shamir sharings $[x]_t + [a]_t$ and $[y]_t + [b]_t$. Observe that, if we set $u = x + a$ and $v = y + b$, the above equation allows us to locally compute a degree-t Shamir sharing of $z := (u - a) \cdot (v - b)$ using a Beaver triple $([a]_t, [b]_t, [c]_t)$. In particular, the values u, v *can be learnt after* preparing the Beaver triple. For multiplications in the second layer, our idea is to transform each input sharing to the form of $u - [a]_t$, where u is a public element and $[a]_t$ is a degree-t Shamir sharing. We refer to this form as the *Beaver-triple friendly form*. Moreover, the sharing $[a]_t$ is known to all parties *before evaluating the first layer*. In this way, for an multiplication gate in the second layer with input sharings $u - [a]_t$ and $v - [b]_t$, we can prepare the Beaver triple $([a]_t, [b]_t, [c]_t)$ *in parallel with* the multiplications in the first layer.

We note that an input sharing of a multiplication gate in the second layer may come from three places:

- This sharing is an input sharing of the circuit.
- This sharing is an output sharing of an addition gate in the first layer.
- This sharing is an output sharing of a multiplication gate in the first layer.

Note that an addition gate can be evaluated without interaction. For the first two cases, all parties can locally compute this sharing. Let $[x]_t$ denote such a sharing. Note that $[x]_t = 0 - (-[x]_t)$ is already in the Beaver-triple friendly form, and $(-[x]_t)$ is known before evaluating the first layer. For the third case, we want the output sharing of a multiplication gate in the first layer to have the Beaver-triple friendly form $u - [a]_t$, and $[a]_t$ is known before evaluating this gate. We note that the original multiplication protocol in [28] satisfies our requirement. Recall that in the original multiplication protocol in [28]:

1. $P_{\texttt{king}}$ reconstructs a degree-$2t$ Shamir sharing $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$ and sends e to other parties.
2. All parties locally compute $[z]_t := e - [r]_t$.

In particular, the random double sharings $([r]_t, [r]_{2t})$ are prepared before evaluating this gate.

In summary, a two-layer circuit can be evaluated as follows:

- For each input sharing in the second layer, all parties transform it to the Beaver-triple friendly form, denoted by $u - [a]_t$, such that $[a]_t$ is known to all parties.
- For each multiplication gate in the first layer, suppose $[x]_t, [y]_t$ are the input sharings. All parties use the original multiplication protocol in [28] to compute $[z]_t$, where $z = x \cdot y$. For each multiplication gate in the second layer, suppose $u - [a]_t, v - [b]_t$ are the input sharings. All parties use our multiplication protocol MULT on $[a]_t, [b]_t$ to compute $[c]_t$, where $c = a \cdot b$. Note that these two kinds of multiplications can be computed in parallel.
- For each multiplication gate in the second layer, suppose $u - [a]_t, v - [b]_t$ are the input sharings. Note that we have learnt u, v when evaluating the first layer, and we have computed the Beaver triple $([a]_t, [b]_t, [c]_t)$. Therefore, all parties compute $[z]_t := u \cdot v - u \cdot [b]_t - v \cdot [a]_t + [c]_t$.

We note that the original multiplication protocol in [28] requires the communication of 6 elements per party. Next, we show how to reduce the communication cost to 5 elements without breaking the form of the output sharing.

6.3.2 Improving the Original Multiplication Protocol in [28]

Recall that in the original multiplication protocol in [28]:

1. $P_{\texttt{king}}$ reconstructs a degree-$2t$ Shamir sharing $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$ and sends e to other parties.
2. All parties locally compute $[z]_t := e - [r]_t$.

To keep the form of the output sharing, $P_{\texttt{king}}$ cannot replace e by a degree-t Shamir sharing $[e]_t$. Furthermore, to protect the secrecy of the multiplication result $x \cdot y$, r needs to be uniformly random. Our main observation is that r being uniform is not equivalent to the double sharings $([r]_t, [r]_{2t})$ being uniform. To this end, we first decouple the relation between r and $([r]_t, [r]_{2t})$. Note that a pair of double sharings $([r]_t, [r]_{2t})$ is equivalent to a pair of sharings $([r]_t, [o]_{2t})$, where the first sharing is a degree-t Shamir sharing of r and the second sharing is a degree-$2t$ Shamir sharing of $o = 0$. To see this, given $([r]_t, [r]_{2t})$, we can set $[o]_{2t} := [r]_{2t} - [r]_t$; given $([r]_t, [o]_{2t})$, we can set $[r]_{2t} := [r]_t + [o]_{2t}$. When using a pair of sharings $([r]_t, [o]_{2t})$, the multipliation protocol becomes:

1. All parties locally compute $[e]_{2t} := [x]_t \cdot [y]_t + [r]_t + [o]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[e]_{2t}$ and reconstructs the secret e. Then $P_{\texttt{king}}$ sends the value e to all other parties.
3. After receiving e from $P_{\texttt{king}}$, all parties locally compute $[z]_t := e - [r]_t$.

Note that $[o]_{2t}$ is only used to compute $[e]_{2t}$. When $P_{\texttt{king}}$ is an honest party, $[o]_{2t}$ does not need to be a uniformly random degree-$2t$ sharing of 0. Thus, following the same argument as that in Chapter 6.2, we can use t-wise independent $[o]_{2t}$'s with uniformly random degree-t sharings $[r]_t$'s. In the following, we first introduce a protocol which allows all parties to generate random degree-$2t$ Shamir sharings of 0. Then, we will expand t such sharings into n sharings with t-wise independence.

Preparing Random Degree-$2t$ Shamir Sharings of 0. The functionality $\mathcal{F}_{\text{zero}}$ appears in Functionality 6.8. We follow the idea of preparing random degree-t sharings in [28]. At a high-level, each party generates and distributes a random degree-$2t$ sharing of 0. Then, use the transpose of a Vandermonde matrix acting as a randomness extractor to obtain $t + 1$ random degree-$2t$ sharings of 0 from the n sharings generated by each party. Let $\boldsymbol{M}^{\mathrm{T}}$ be a predetermined and fixed Vandermonde matrix of size $n \times (t + 1)$ (therefore $\boldsymbol{M}$ is a $(t + 1) \times n$ matrix). The protocol ZERO appears in Protocol 6.9. The communication complexity of ZERO is $O(n^2)$ field elements. The amortized cost per sharing is 2 elements per party.

Lemma 6.2. *The protocol* ZERO *securely computes the functionality $\mathcal{F}_{zero}$ in the presence of a fully malicious adversary controlling t corrupted parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Recall that $\mathcal{C}orr$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation for ZERO. In the first step, when an honest party P_i needs to distribute a random degree-$2t$ sharing $[o^{(i)}]_{2t}$ of $o^{(i)} = 0$, for each corrupted party P_j, $\mathcal{S}$ samples a random element

Figure 6.8: Functionality $\mathcal{F}_{\text{zero}}$

1. $\mathcal{F}_{\text{zero}}$ receives from the adversary the set of shares $\{r_i\}_{i\in\mathcal{C}orr}$.
2. $\mathcal{F}_{\text{zero}}$ randomly samples t elements as the shares of the first t honest parties. Based on the secret $o = 0$, the t shares of the first t honest parties, and the t shares $\{r_i\}_{i\in\mathcal{C}orr}$ of corrupted parties, $\mathcal{F}_{\text{zero}}$ reconstructs the whole sharing $[o]_{2t}$. $\mathcal{F}_{\text{zero}}$ distributes the shares of $[o]_{2t}$ to honest parties.

Figure 6.9: Protocol ZERO

1. Each party P_i randomly samples a degree-$2t$ Shamir sharing of 0, denoted by $[o^{(i)}]_{2t}$ and distributes the shares to other parties.
2. All parties agree on a Vandermonde matrix $\boldsymbol{M}^{\mathrm{T}}$ of size $n \times (t+1)$. Then $\boldsymbol{M}$ is a matrix of size $(t+1) \times n$. All parties locally compute

$$([\tilde{o}^{(1)}]_{2t}, [\tilde{o}^{(2)}]_{2t}, \ldots, [\tilde{o}^{(t+1)}]_{2t})^{\mathrm{T}} \quad = \quad \boldsymbol{M}([o^{(1)}]_{2t}, [o^{(2)}]_{2t}, \ldots, [o^{(n)}]_{2t})^{\mathrm{T}}$$

   and output $[\tilde{o}^{(1)}]_{2t}, [\tilde{o}^{(2)}]_{2t}, \ldots, [\tilde{o}^{(t+1)}]_{2t}$.

as its share of $[o^{(i)}]_{2t}$ and sends it to the adversary. For each corrupted party P_i, $\mathcal{S}$ receives the shares of $[o^{(i)}]_{2t}$ held by honest parties. $\mathcal{S}$ samples a random degree-$2t$ sharing based on the secret $o^{(i)} = 0$ and the shares held by honest parties, and views this degree-$2t$ sharing as the one distributed by P_i.

In the second step, $\mathcal{S}$ computes the shares of $[\tilde{o}^{(i)}]_{2t}$ held by corrupted parties and passes these shares to $\mathcal{F}_{\text{zero}}$.

Hybrid Arguments. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ changes the way of preparing a random degree-$2t$ Shamir sharing $[o^{(i)}]_{2t}$ of $o^{(i)} = 0$ for each honest party P_i:

1. $\mathcal{S}$ first randomly samples the shares of $[o^{(i)}]_{2t}$ held by corrupted parties and sends them to corrupted parties.
2. Then, based on the secret $o^{(i)} = 0$ and the shares of $[o^{(i)}]_{2t}$ held by corrupted parties, $\mathcal{S}$ samples a random degree-$2t$ sharing $[o^{(i)}]_{2t}$ (in the same way as $\mathcal{F}_{\text{zero}}$ in Step 2). Then $\mathcal{S}$ distributes the shares to the rest of honest parties.

For each corrupted party P_i, $\mathcal{S}$ locally computes the shares of $[o^{(i)}]_{2t}$ held by corrupted parties.

Note that this does not change the distribution of the random sharings generated by honest parties. The distribution of **Hybrid**$_0$ is identical to the distribution of **Hybrid**$_1$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ omits the second step when preparing $[o^{(i)}]_{2t}$ for each honest party P_i in **Hybrid**$_1$. Recall that in **Hybrid**$_1$, for all $i \in [n]$, $\mathcal{S}$ has computed the shares of $[o^{(i)}]_{2t}$ held by corrupted parties. For each $[\tilde{o}^{(i)}]_{2t}$, $\mathcal{S}$ computes the shares of corrupted parties and sends them to $\mathcal{F}_{\text{zero}}$.

We show that the distribution of **Hybrid**$_2$ is identical to the distribution of **Hybrid**$_1$.

Let $\boldsymbol{M}^{\mathcal{H}}$ denote the sub-matrix of $\boldsymbol{M}$ containing the columns of $\boldsymbol{M}$ with indices in $\mathcal{H}$ and $\boldsymbol{M}^{\mathcal{C}orr}$ denote the sub-matrix of $\boldsymbol{M}$ containing the columns of $\boldsymbol{M}$ with indices in $\mathcal{C}orr$. Let $([o^{(i)}]_{2t})_{\mathcal{H}}$ denote the vector of the sharings dealt by parties in $\mathcal{H}$ and $([o^{(i)}]_{2t})_{\mathcal{C}orr}$ denote the vector of the sharings dealt by parties in $\mathcal{C}orr$. Then,

$$\begin{aligned}([\tilde{o}^{(1)}]_{2t}, [\tilde{o}^{(2)}]_{2t}, \ldots, [\tilde{o}^{(t+1)}]_{2t})^{\mathrm{T}} &= \boldsymbol{M}([o^{(1)}]_{2t}, [o^{(2)}]_{2t}, \ldots, [o^{(n)}]_{2t})^{\mathrm{T}} \\ &= \boldsymbol{M}^{\mathcal{H}}([o^{(i)}]_{2t})_{\mathcal{H}}^{\mathrm{T}} + \boldsymbol{M}^{\mathcal{C}orr}([o^{(i)}]_{2t})_{\mathcal{C}orr}^{\mathrm{T}}\end{aligned}$$

Note that $\boldsymbol{M}^{\mathcal{H}}$ is a $(t+1) \times (t+1)$ matrix. By the property of Vandermonde matrices, $\boldsymbol{M}^{\mathcal{H}}$ is invertible. Therefore, given the sharings $\{[o^{(i)}]_{2t}\}_{i \in \mathcal{C}orr}$ dealt by corrupted parties, there is a one-to-one map from $\{[o^{(i)}]_{2t}\}_{i \in \mathcal{H}}$ to $\{[\tilde{o}^{(i)}]_{2t}\}_{i \in [t+1]}$. Note that the only difference between **Hybrid**$_1$ and **Hybrid**$_2$ is that, in **Hybrid**$_1$, $\{[o^{(i)}]_{2t}\}_{i \in \mathcal{H}}$ are randomly generated (based on the shares which have been sent to corrupted parties) while in **Hybrid**$_2$, $\mathcal{F}_{\text{zero}}$ directly generates $\{[\tilde{o}^{(i)}]_{2t}\}_{i \in [t+1]}$ based on the shares that corrupted parties should hold. However, this does not change the distribution of the shares of $\{[\tilde{o}^{(i)}]_{2t}\}_{i \in [t+1]}$ held by honest parties. To see this, note that for any sharings $\{[\tilde{o}^{(i)}]_{2t}\}_{i \in [t+1]}$ generated by $\mathcal{F}_{\text{zero}}$, we can compute back to a set of valid sharings $\{[o^{(i)}]_{2t}\}_{i \in \mathcal{H}}$. Therefore, the distribution of **Hybrid**$_2$ is identical to the distribution of **Hybrid**$_1$.

Note that **Hybrid**$_2$ is the execution in the ideal world and the distribution of **Hybrid**$_2$ is identical to the distribution of **Hybrid**$_0$, the execution in the real world. □

The Improved Multiplication Protocol. For a sequence of n multiplication gates, all parties first prepare n random degree-t Shamir sharings using $\mathcal{F}_{\text{rand}}$, denoted by

$$[r^{(1)}]_t, \ldots, [r^{(n)}]_t.$$

Recall that the amortized communication cost of the instantiation of $\mathcal{F}_{\text{rand}}$ in [28, 42] is 2 elements per sharing per party. Then, all parties invoke $\mathcal{F}_{\text{zero}}$ to prepare t random degree-$2t$ Shamir sharings of 0, denoted by

$$[o^{(1)}]_t, \ldots, [o^{(t)}]_t.$$

These t sharings are expanded to n sharings with t-wise independence. As EXPAND, we will use a predetermined $n \times t$ Vandermonde matrix $\boldsymbol{M}$. The protocol EXPANDZERO appears in Protocol 6.10.

For the i-th multiplication gate, we will use $([r^{(i)}]_t, [\tilde{o}^{(i)}]_{2t}, P_i)$ and P_i will act as $P_{\texttt{king}}$. The protocol IMPROVED-DN-MULT appears in Protocol 6.11. As for the amortized communication cost per gate:

- Preparing one random degree-t Shamir sharing using $\mathcal{F}_{\text{rand}}$ requires to communicate 2 elements per party.

Figure 6.10: Protocol EXPANDZERO

1. All parties agree on an $n \times t$ hyper-intertible matrix $\boldsymbol{M}$. All parties locally compute

$$([\tilde{o}^{(1)}]_{2t}, \ldots, [\tilde{o}^{(n)}]_{2t})^{\mathrm{T}} \quad = \quad \boldsymbol{M}([o^{(1)}]_{2t}, \ldots, [o^{(t)}]_{2t})^{\mathrm{T}}$$

2. All parties output $\{([\tilde{o}^{(i)}]_{2t}, P_i)\}_{i=1}^{n}$, where $([\tilde{o}^{(i)}]_{2t}, P_i)$ will be used for a multiplication gate handled by P_i.

- Preparing one t-wise independent degree-$2t$ Shamir sharing of 0 using $\mathcal{F}_{\text{zero}}$ and EXPANDZERO requires to communicate $2 \cdot t/n$ elements per party.
- The protocol IMPROVED-DN-MULT requires to communicate 2 elements per party.

In summary, the amortized communication cost per gate is 5 elements per party.

Figure 6.11: Protocol IMPROVED-DN-MULT

1. Let $([r]_t, [o]_{2t}, P_i)$ be the random sharings which will be used in the protocol. Let $[x]_t, [y]_t$ denote the input sharings.
2. All parties locally compute $[e]_{2t} = [x]_t \cdot [y]_t + [r]_t + [o]_{2t}$.
3. P_i collects all shares and reconstructs the secret $e = x \cdot y + r$. Then P_i sends e to other parties.
4. All parties locally compute $[z]_t = e - [r]_t$.

6.3.3 Evaluating a Two-Layer Circuit

Given a two-layer circuit, we assume that all parties hold a degree-t Shamir sharing for each input wire in the beginning. As described above, we will use IMPROVED-DN-MULT to evaluate multiplication gates in the first layer. For multiplication gates in the second layer, note that all parties only need to obtain the output sharings. Therefore, we can use MULT, which only requires 4 elements per gate per party, to evaluate multiplication gates in the second layer.

Suppose there are N_1 multiplication gates in the first layer, and N_2 multiplication gates in the second layer. We assume that all parties have prepared the correlated randomness associated with these multiplication gates, i.e., N_1 pairs of sharings in the form of $([r]_t, [o]_t, P_i)$, and N_2 pairs of sharings in the form of $([r]_t, [r]_{2t}, P_i)$. In the main protocol, these sharings are prepared together at the beginning of the protocol. Then all parties execute EVALUATE (Protocol 6.12) to compute the output sharings of this circuit.

Figure 6.12: Protocol EVALUATE

1. All parties start with holding a degree-t Shamir sharing for each input wire of this circuit. For each multiplication gate in the second layer, we will transform the input sharings to the Beaver-triple friendly form $u - [a]_t$. Consider the following three cases.
   - If this sharing is an input sharing of the circuit, denoted by $[x]_t$, all parties set $u := 0$ and $[a]_t := -[x]_t$.
   - If this sharing is an output sharing of an addition gate in the first layer, all parties first locally compute this sharing, denoted by $[x]_t$, and then set $u := 0$ and $[a]_t := -[x]_t$.
   - If this sharing is an output sharing of a multiplication gate in the first layer, suppose $([r]_t, [o]_{2t}, P_i)$ are associated with this gate. All parties set $[a]_t := [r]_t$. The value u, which corresponds to e in IMPROVED-DN-MULT, will be computed when this multiplication gate is evaluated.
2. For each multiplication gate with input sharings $[x]_t, [y]_t$ in the first layer, all parties invoke IMPROVED-DN-MULT to compute $[z]_t$ where $z := x \cdot y$. For each multiplication gate with input sharings $(u - [a]_t), (v - [b]_t)$ in the second layer, where all parties have learnt the sharings $[a]_t, [b]_t$, all parties invoke MULT to compute $[c]_t$ where $c := a \cdot b$.
3. For each multiplication gate in the first layer, let e be the reconstruction result distributed by $P_{\texttt{king}}$ in IMPROVED-DN-MULT. If the output sharing of this gate is used as an input sharing of a multiplication gate in the second layer, all parties set $u := e$ for this input sharing.
4. Finally, for each multiplication gate with input sharings $(u - [a]_t), (v - [b]_t)$ in the second layer, all parties locally compute

   $$[z]_t := u \cdot v - u \cdot [b]_t - v \cdot [a]_t + [c]_t$$

   as the output sharing of this gate.

6.3.4 Main Protocol

Now we are ready to present the main protocol. Recall that we are in the client-server model. In particular, all the inputs belong to the clients, and only the clients receive the outputs. The functionality $\mathcal{F}_{\text{main}}$ appears in Functionality 6.13.

As the semi-honest DN protocol [28], our protocol includes 3 phases:

- Input Phase: The clients will share their inputs to the parties.
- Evaluation Phase: The whole circuit will be partitioned into a sequence of two-layer sub-circuits. We will evaluate each sub-circuit using EVALUATE.
- Output Phase: All parties reconstruct the outputs to the clients.

The protocol MAIN-ROUND appears in Protocol 6.14.

Figure 6.13: Functionality $\mathcal{F}_{\text{main}}$

1. $\mathcal{F}_{\text{main}}$ receives from all clients their inputs.
2. $\mathcal{F}_{\text{main}}$ evaluates the circuit and computes the output. $\mathcal{F}_{\text{main}}$ distributes the output to all clients.

Figure 6.14: Protocol MAIN-ROUND

1. **Input Phase**:

   Let $\mathsf{Client}_1, \ldots, \mathsf{Client}_{\mathsf{c}}$ denote the clients who provide inputs. For every input gate of Client_i, Client_i samples a random degree-t Shamir sharing of its input, $[x]_t$, and distributes the shares to all parties.

2. **Evaluation Phase – Preparing Correlated Randomness**:

   All parties start with holding a degree-t sharing for each input gate. The circuit is partitioned into a sequence of two-layer sub-circuits. Let N_1 denote the number of multiplications in the first layer of all sub-circuits, and N_2 denote the number of multiplications in the second layer of all sub-circuits. All parties prepare the correlated randomness as follows:

   - All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare N_1 random degree-t Shamir sharings. Then all parties invoke $\mathcal{F}_{\text{zero}}$ to prepare $N_1 \cdot t/n$ random degree-$2t$ Shamir sharings of 0, and invoke EXPANDZERO to obtain N_1 degree-$2t$ Shamir sharings of 0. These sharings are transformed to N_1 pairs of sharings in the form of $([r]_t, [o]_{2t}, P_i)$.
   - All parties invoke $\mathcal{F}_{\text{doubleRand}}$ to prepare $N_2 \cdot t/n$ pairs of random double sharings. Then all parties invoke EXPAND to obtain N_2 pairs of double sharings in the form of $([r]_t, [r]_{2t}, P_i)$.

3. **Evaluation Phase – Evaluating Two-Layer Circuits**:

   All sub-circuits are evaluated in a predetermined topological order. For each sub-circuit with all the input sharings prepared, all parties invoke EVALUATE to compute the output sharings.

4. **Output Phase**:

   For each output gate of Client_i, let $[x]_t$ denote the input sharing. All parties send their shares of $[x]_t$ to Client_i to let Client_i reconstruct x.

Lemma 6.3. *Let* c *be the number of clients and* $n = 2t+1$ *be the number of parties. The protocol* MAIN-ROUND *securely computes the ideal functionality* $\mathcal{F}_{main}$ *in the* $\{\mathcal{F}_{rand}, \mathcal{F}_{zero}, \mathcal{F}_{doubleRand}, \mathcal{F}_{coin}, \mathcal{F}_{multVerify}\}$*-hybrid model in the presence of a semi-honest adversary controlling up to* c *clients and* t *parties.*

Proof. As we argued in Chapter 2, it is sufficient to focus on adversaries who control exactly t

parties. The correctness of MAIN-ROUND follows from the description.

Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties and honest clients. Recall that $\mathcal{C}orr$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation for MAIN-ROUND. We describe the strategy of $\mathcal{S}$ phase by phase.

- Simulation for Input Phase:
  For an input x belongs to an honest client, $\mathcal{S}$ randomly samples t elements as the shares of $[x]_t$ of corrupted parties. Then $\mathcal{S}$ sends these shares to corrupted parties.
  For an input x belongs to a corrupted client, $\mathcal{S}$ receives from the adversary the shares held by honest parties. Note that, $\mathcal{S}$ learns $t+1$ shares of $[x]_t$. $\mathcal{S}$ reconstructs the whole sharing $[x]_t$ and sends x as the input of this corrupted client to $\mathcal{F}_{\text{main}}$.
  Note that for each input sharing, $\mathcal{S}$ learns the shares held by corrupted parties.
- Simulation for Evaluation Phase – Preparing Correlated Randomness:
  In this part, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{zero}}, \mathcal{F}_{\text{doubleRand}}$ and receives the shares of corrupted parties. $\mathcal{S}$ follows the protocol EXPANDZERO and EXPAND to compute the shares of corrupted parties.
- Simulation for Evaluation Phase – Evaluating Two-Layer Circuits:
  In this part, $\mathcal{S}$ simulates the behaviors of honest parties in EVALUATE for each sub-circuit. For each sub-circuit with all the input sharings prepared, $\mathcal{S}$ will know the shares held by corrupted parties. Note that this is true for the first sub-circuit. We describe the strategy of $\mathcal{S}$ step by step.
  - In the first step, $\mathcal{S}$ follows the protocol to compute the shares of corrupted parties for the input sharings of each multiplication gate in the second layer.
  - In the second step, $\mathcal{S}$ simulates IMPROVED-DN-MULT and MULT as follows. For IMPROVED-DN-MULT:
    - Let $[x]_t, [y]_t$ denote the input sharings. $\mathcal{S}$ has computed the shares of $[x]_t, [y]_t$ held by corrupted parties. Let $([r]_t, [o]_{2t}, P_i)$ denote the sharings associated with this gate. $\mathcal{S}$ learns the shares of $[r]_t, [o]_{2t}$ held by corrupted parties when emulating $\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{zero}}$ and simulating EXPANDZERO.
    - $\mathcal{S}$ computes the shares of $[e]_{2t} := [x]_t \cdot [y]_t + [r]_t + [o]_{2t}$ held by corrupted parties. If P_i is corrupted, $\mathcal{S}$ samples random elements as the shares of $[e]_{2t}$ of honest parties.
    - Depending on whether P_i is honest, there are two cases:
      - If P_i is honest, $\mathcal{S}$ receives the shares from corrupted parties. $\mathcal{S}$ samples a random element as e and sends e to corrupted parties.
      - If P_i is corrupted, $\mathcal{S}$ sends the shares of $[e]_{2t}$ of honest parties to P_i. $\mathcal{S}$ receives e from P_i.
    - In the last step, $\mathcal{S}$ computes the shares of $[z]_t := e - [r]_t$ held by corrupted parties.

    For MULT:

- Let $[x]_t, [y]_t$ denote the input sharings. $\mathcal{S}$ has computed the shares of $[x]_t, [y]_t$ held by corrupted parties. Let $([r]_t, [r]_{2t}, P_i)$ denote the sharings associated with this gate. $\mathcal{S}$ learns the shares of $[r]_t, [r]_{2t}$ held by corrupted parties when emulating $\mathcal{F}_{\text{doubleRand}}$ and simulating EXPAND.
- $\mathcal{S}$ computes the shares of $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$ held by corrupted parties. If P_i is corrupted, $\mathcal{S}$ samples random elements as the shares of $[e]_{2t}$ of honest parties.
- Depending on whether P_i is honest, there are two cases:
  - If P_i is honest, $\mathcal{S}$ receives the shares from corrupted parties. For $[e]_t$, $\mathcal{S}$ samples random elements as the shares of corrupted parties.
  - If P_i is corrupted, $\mathcal{S}$ sends the shares of $[e]_{2t}$ of honest parties to P_i. Then, $\mathcal{S}$ receives the shares of $[e]_t$ of honest parties from P_i and computes the shares held by corrupted parties.
- In the last step, $\mathcal{S}$ computes the shares of $[z]_t := [e]_t - [r]_t$ held by corrupted parties.

▪ In the rest of two steps (which only contain local computation), $\mathcal{S}$ follows the protocol and computes the shares of corrupted parties for each degree-t Shamir sharing.

- Simulation for Output Phase:
  For each output gate with $[x]_t$ associated with it, if the receiver is an honest client, $\mathcal{S}$ receives from the adversary the shares held by corrupted parties.
  If the receiver is a corrupted client, $\mathcal{S}$ receives the result x from $\mathcal{F}_{\text{main}}$. Then, based on the shares of $[x]_t$ held by corrupted parties and the secret x, $\mathcal{S}$ reconstructs the shares held by honest parties, and sends these shares to the adversary.

Hybrid Arguments. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ computes the input of corrupted clients and sends them to $\mathcal{F}_{\text{main}}$. The distribution of **Hybrid**$_1$ is identical to **Hybrid**$_0$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ simulates the output phase as described above. The only difference is that, for a corrupted client, the shares of honest parties are computed from the secret received from $\mathcal{F}_{\text{main}}$ and the shares held by corrupted parties. Note that for a degree-t Shamir sharing, the shares of honest parties are fully determined by the secret and the shares of corrupted parties. By the correctness of MAIN-ROUND, the secret received from $\mathcal{F}_{\text{main}}$ is identical to that in the real world. Therefore, the shares prepared by $\mathcal{S}$ are identical to the real shares held by honest parties.

Therefore, the distribution of **Hybrid**$_2$ is identical to the distribution of **Hybrid**$_1$.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ simulates EVALUATE. Since the parts which require interaction are IMPROVED-DN-MULT and MULT, we only focus on the simulation of these two protocols. We argue the following two points:

- When a corrupted party P_i behaves as $P_{\texttt{king}}$, the shares of $[e]_{2t}$ of honest parties are uniformly random in both IMPROVED-DN-MULT and MULT.
- The value e distributed by $P_{\texttt{king}}$ is uniformly random in IMPROVED-DN-MULT.

For the first point, following from a similar argument in Lemma 6.1, the random sharing $[r]_t + [o]_{2t}$ in IMPROVED-DN-MULT and the random sharing $[r]_{2t}$ in MULT satisfy that the shares of honest parties are uniformly random when $P_{\texttt{king}}$ is a corrupted party. Therefore, the shares of $[e]_{2t}$ of honest parties are uniformly random. For the second point, since we use a uniformly random $[r]_t$ in each IMPROVED-DN-MULT, the secret $e = x \cdot y + r$ is uniformly random.

Note that the differences between $\mathbf{Hybrid}_3$ and $\mathbf{Hybrid}_2$ are that:

- When a corrupted party P_i behaves as $P_{\texttt{king}}$, $\mathcal{S}$ uses uniformly random field elements as the shares of $[e]_{2t}$ of honest parties in both IMPROVED-DN-MULT and MULT.
- When an honest party P_i behaves as $P_{\texttt{king}}$, $\mathcal{S}$ distributes a uniformly random element as e in IMPROVED-DN-MULT and uses uniformly random field elements as the shares of $[e]_t$ of corrupted parties in MULT.

Following from the above two points and the property of degree-t Shamir sharings, the distribution of $\mathbf{Hybrid}_3$ is identical to the distribution of $\mathbf{Hybrid}_2$.

$\mathbf{Hybrid}_4$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{doubleRand}}, \mathcal{F}_{\text{zero}}$ and simulates EXPANDZERO and EXPAND as described above. Note that only the shares of corrupted parties are used in $\mathbf{Hybrid}_3$. Therefore, the distribution of $\mathbf{Hybrid}_4$ is identical to the distribution of $\mathbf{Hybrid}_3$.

$\mathbf{Hybrid}_5$: In this hybrid, $\mathcal{S}$ simulates the input phase. The only difference is that, in $\mathbf{Hybrid}_4$, $\mathcal{S}$ uses the real input of honest clients to generate the input sharings, while in $\mathbf{Hybrid}_5$, $\mathcal{S}$ simply samples random elements as the shares of corrupted parties. Note that the distributions of the shares of corrupted parties in both hybrids are the same. Therefore, the distribution of $\mathbf{Hybrid}_5$ is the same as $\mathbf{Hybrid}_4$.

Note that $\mathbf{Hybrid}_5$ is the execution in the ideal world, and the distribution of $\mathbf{Hybrid}_5$ is identical to the distribution of $\mathbf{Hybrid}_0$, the execution in the real world. □

Analysis of the Concrete Efficiency. In MAIN-ROUND, all multiplication gates in the first layer of all sub-circuits are evaluated by IMPROVED-DN-MULT, which requires 5 elements per party per gate. All multiplication gates in the second layer of all sub-circuits are evaluated by MULT, which requires 4 elements per party per gate. Assuming that the number of multiplication gates in the first layer is roughly the same as the number of multiplication gates in the second layer, the concrete efficiency of MAIN-ROUND is 4.5 elements per party per gate. Note that each sub-circuit is evaluated within one round of multiplication. Therefore, we reduce the number of rounds by a factor of 2. The overall communication complexity is $O(|C| \cdot n + n^2)$ field elements.

Theorem 4.2. *Let n be a positive integer and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n + 1$. For all arithmetic circuit C over $\mathbb{F}$, there is an information-theoretic n-party MPC protocol, which achieves semi-honest security against a semi-honest adversary controlling $t < n/2$ corrupted parties, with communication complexity $O(|C| \cdot n + n^2)$ elements, where $|C|$ is the circuit size. Furthermore, the concrete efficiency is 4.5 elements per party per multiplication gate but halving the number of rounds.*

Chapter 7

Fast Verification for a Batch of Multiplication Tuples

In this chapter, we introduce our new method to efficiently verify a batch of multiplication tuples. In [38], Genkin et al. showed that the semi-honest DN protocol is secure up to an additive attack in the presence of a fully malicious adversary. An additive attack means that the adversary is able to change the multiplication result by adding an arbitrary fixed value. As one corollary, the DN protocol provides full privacy of honest parties before reconstructing the output. Therefore, a straightforward strategy to achieve security-with-abort is to (1) run the semi-honest DN protocol till the output phase, (2) check the correctness of the computation, and (3) reconstruct the output only if the check passes.

Following from [24], we model the additive attack in the functionality $\mathcal{F}_{\text{mult-mal}}$, which takes two degree-t Shamir sharings $[x]_t, [y]_t$ and outputs the multiplication result $[x \cdot y]_t$. The description of $\mathcal{F}_{\text{mult-mal}}$ can be found in Functionality 7.1.

Figure 7.1: Functionality $\mathcal{F}_{\text{mult-mal}}$

1. Let $[x]_t, [y]_t$ denote the input sharings. $\mathcal{F}_{\text{mult-mal}}$ receives from honest parties their shares of $[x]_t, [y]_t$. Then $\mathcal{F}_{\text{mult-mal}}$ reconstructs the secrets x, y. $\mathcal{F}_{\text{mult-mal}}$ further computes the shares of $[x]_t, [y]_t$ held by corrupted parties, and sends these shares to the adversary.
2. $\mathcal{F}_{\text{mult-mal}}$ receives from the adversary a value d and a set of shares $\{z_i\}_{i \in \mathcal{C}orr}$.
3. $\mathcal{F}_{\text{mult-mal}}$ computes $x \cdot y + d$. Based on the secret $z := x \cdot y + d$ and the t shares $\{z_i\}_{i \in \mathcal{C}orr}$, $\mathcal{F}_{\text{mult-mal}}$ reconstructs the whole sharing $[z]_t$ and distributes the shares of $[z]_t$ to honest parties.

Since $\mathcal{F}_{\text{mult-mal}}$ does not guarantee the correctness of the multiplications, all parties need to verify the multiplication tuples computed by $\mathcal{F}_{\text{mult-mal}}$ at the end of the protocol. The functionality $\mathcal{F}_{\text{multVerify}}$ takes N multiplication tuples as input and outputs to all parties a single bit b indicating whether all multiplication tuples are correct. The description of $\mathcal{F}_{\text{multVerify}}$ can be found in

Functionality 7.2.

Figure 7.2: Functionality $\mathcal{F}_{\text{multVerify}}$

1. Let N denote the number of multiplication tuples. The multiplication tuples are denoted by

$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(N)}]_t.[y^{(N)}]_t, [z^{(N)}]_t).$$

2. For all $i \in [N]$, $\mathcal{F}_{\text{multVerify}}$ receives from honest parties their shares of $[x^{(i)}]_t$, $[y^{(i)}]_t$, $[z^{(i)}]_t$. Then $\mathcal{F}_{\text{multVerify}}$ reconstructs the secrets $x^{(i)}, y^{(i)}, z^{(i)}$. $\mathcal{F}_{\text{multVerify}}$ further computes the shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$ held by corrupted parties and sends these shares to the adversary.
3. For all $i \in [N]$, $\mathcal{F}_{\text{multVerify}}$ computes $d^{(i)} = z^{(i)} - x^{(i)} \cdot y^{(i)}$ and sends $d^{(i)}$ to the adversary.
4. Finally, let $b \in \{\texttt{abort}, \texttt{accept}\}$ denote whether there exists $i \in [N]$ such that $d^{(i)} \neq 0$. $\mathcal{F}_{\text{multVerify}}$ sends b to the adversary and waits for its response.
   - If the adversary replies `continue`, $\mathcal{F}_{\text{multVerify}}$ sends b to honest parties.
   - If the adversary replies `abort`, $\mathcal{F}_{\text{multVerify}}$ sends `abort` to honest parties.

With $\mathcal{F}_{\text{multVerify}}$, we can compile the semi-honest DN protocol to a maliciously secure one (see more discussion in Chapter 7.4). Our main contribution is an efficient verification protocol that realizes $\mathcal{F}_{\text{multVerify}}$ with sub-linear communication complexity in the number of multiplication tuples[1]. We first review two building blocks from previous works.

7.1 Extension of the DN Multiplication Protocol

In this part, we review a natural extension to the DN Multiplication Protocol [28]. Recall that the original DN multiplication protocol uses a pair of random double sharings for each multiplication gate. The idea is to locally compute a degree-$2t$ Shamir sharing of the multiplication result, and then ask $P_{\texttt{king}}$ to do degree reduction. In [38], Genkin et al. prove that the semi-honest DN protocol is secure up to an additive attack in the presence of a fully malicious adversary.

Lemma 7.1 ([38])**.** *The protocol* DN-MULT *securely computes the functionality $\mathcal{F}_{\textit{mult-mal}}$ in the $\mathcal{F}_{\textit{doubleRand}}$-hybrid model in the presence of a fully malicious adversary controlling t corrupted parties.*

In essence, the DN Multiplication Protocol does a degree reduction from $[x \cdot y]_{2t} = [x]_t \cdot [y]_t$ to $[x \cdot y]_t$. For two vectors of degree-t Shamir sharings $([x_1]_t, \ldots, [x_\ell]_t)$ and $([y_1]_t, \ldots, [y_\ell]_t)$, to compute $[z]_t = [\sum_{j=1}^{\ell} x_j \cdot y_j]_t$, we can first compute $[\sum_{j=1}^{\ell} x_j \cdot y_j]_{2t} = \sum_{j=1}^{\ell} [x_j]_t \cdot [y_j]_t$ and then use the same idea as the DN Multiplication Protocol to compute $[\sum_{j=1}^{\ell} x_j \cdot y_j]_t$ from

[1]We note that a concurrent work [17] also realizes Functionality 7.2 with sub-linear communication complexity in the number of multiplication tuples based on [13].

$[\sum_{j=1}^{\ell} x_j \cdot y_j]_{2t}$. In this way, the cost is just one multiplication operation. This idea has been observed in several previous works and in particular, has been used in [24] to design an MPC protocol for a small field.

The description of the functionality $\mathcal{F}_{\text{extendMult}}$ appears in Functionality 7.3, and the description of the extended DN Multiplication Protocol (denoted by EXTEND-MULT) appears in Protocol 7.4. The communication complexity of EXTEND-MULT is $O(n)$ field elements.

Figure 7.3: Functionality $\mathcal{F}_{\text{extendMult}}$

1. Let $([x_1]_t, \ldots, [x_\ell]_t)$ and $([y_1]_t, \ldots, [y_\ell]_t)$ denote the input vectors of sharings. $\mathcal{F}_{\text{extendMult}}$ receives from honest parties their shares of $([x_1]_t, \ldots, [x_\ell]_t)$ and $([y_1]_t, \ldots, [y_\ell]_t)$. Then $\mathcal{F}_{\text{extendMult}}$ reconstructs the secrets $x_1, \ldots, x_\ell$ and $y_1, \ldots, y_\ell$. $\mathcal{F}_{\text{extendMult}}$ further computes the shares of $([x_1]_t, \ldots, [x_\ell]_t)$ and $([y_1]_t, \ldots, [y_\ell]_t)$ held by corrupted parties, and sends these shares to the adversary.
2. $\mathcal{F}_{\text{extendMult}}$ receives from the adversary a value d and a set of shares $\{z_i\}_{i \in \mathcal{C}orr}$.
3. $\mathcal{F}_{\text{extendMult}}$ computes $z = d + \sum_{j=1}^{\ell} x_j \cdot y_j$. Based on the secret z and the t shares $\{z_i\}_{i \in \mathcal{C}orr}$, $\mathcal{F}_{\text{extendMult}}$ reconstructs the whole sharing $[z]_t$ and distributes the shares of $[z]_t$ to honest parties.

Figure 7.4: Protocol EXTEND-MULT

1. All parties agree on a special party $P_{\texttt{king}}$. Let $([x_1]_t, \ldots, [x_\ell]_t)$ and $([y_1]_t, \ldots, [y_\ell]_t)$ denote the input vectors of sharings.
2. All parties invoke $\mathcal{F}_{\text{doubleRand}}$ to prepare a pair of random double sharings $([r]_t, [r]_{2t})$.
3. All parties locally compute $[e]_{2t} = \sum_{j=1}^{\ell} [x_j]_t \cdot [y_j]_t + [r]_{2t}$.
4. $P_{\texttt{king}}$ collects all shares and reconstructs the secret value e. Then $P_{\texttt{king}}$ randomly generates a degree-t sharing $[e]_t$ and distributes the shares to other parties.
5. All parties locally compute $[z]_t = [e]_t - [r]_t$.

Lemma 7.2. *The protocol* EXTEND-MULT *securely computes the functionality $\mathcal{F}_{\text{extendMult}}$ in the $\mathcal{F}_{\text{doubleRand}}$-hybrid model in the presence of a fully malicious adversary controlling t corrupted parties.*

Proof. Recall that $\mathcal{C}orr$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties. Recall that in Lemma 6.1, we have argued that the random degree-$2t$ sharing $[r]_{2t}$ output by $\mathcal{F}_{\text{doubleRand}}$ satisfies that the shares of honest parties are uniformly random, and are independent of the shares chosen by the adversary.

Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties.

Simulation of EXTEND-MULT. In the beginning, $\mathcal{S}$ receives from $\mathcal{F}_{\text{extendMult}}$ the shares of $([x_1]_t, \ldots, [x_\ell]_t)$ and $([y_1]_t, \ldots, [y_\ell]_t)$ held by corrupted parties. When invoking $\mathcal{F}_{\text{doubleRand}}$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{doubleRand}}$ and receives from the adversary the shares of $[r]_t, [r]_{2t}$ held by corrupted parties.

In Step 3, for each honest party, $\mathcal{S}$ samples a random element as its share of $[e]_{2t}$. For each corrupted party, $\mathcal{S}$ computes its share of $[e]_{2t}$. Then $\mathcal{S}$ reconstructs the secret e. Depending on whether $P_{\texttt{king}}$ is an honest party, there are two cases:

- If $P_{\texttt{king}}$ is an honest party, $\mathcal{S}$ receives from corrupted parties their shares of $[e]_{2t}$ (which can be different from the shares computed by $\mathcal{S}$). $\mathcal{S}$ uses these shares and the shares of honest parties to reconstruct the secret e'. Then, $\mathcal{S}$ computes the difference $d := e' - e$. $\mathcal{S}$ randomly generates a degree-t Shamir sharing $[e']_t$ and distributes the shares to corrupted parties.
- If $P_{\texttt{king}}$ is a corrupted party, $\mathcal{S}$ sends the shares of $[e]_{2t}$ of honest parties to $P_{\texttt{king}}$. $\mathcal{S}$ receives from $P_{\texttt{king}}$ the shares of $[e]_t$ held by honest parties. Then $\mathcal{S}$ reconstructs the secret e' and computes the difference $d := e' - e$. $\mathcal{S}$ also computes the shares of $[e]_t$ held by corrupted parties.

In Step 5, $\mathcal{S}$ computes the shares of $[z]_t$ held by corrupted parties. Note that $\mathcal{S}$ has computed the shares of $[e]_t$ held by corrupted parties and received the shares of $[r]_t$ held by corrupted parties when emulating $\mathcal{F}_{\text{doubleRand}}$. Finally, $\mathcal{S}$ sends the difference d and the shares of $[z]_t$ of corrupted parties to $\mathbb{F}mult$.

Hybrids Argument. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ computes the difference d and the shares of $[z]_t$ held by corrupted parties as described above. $\mathcal{S}$ sends d and the shares of $[z]_t$ of corrupted parties to $\mathcal{F}_{\text{extendMult}}$. Each honest party uses the share received from $\mathcal{F}_{\text{extendMult}}$ instead of the real share.

Note that the shares of $[z]_t$ held by honest parties are determined by the shares of corrupted parties and the secret, which is determined by the inner-product result $\sum_{j=1}^{\ell} x_j \cdot y_j$ and the difference d. Therefore, the distribution of **Hybrid**$_1$ is the same as **Hybrid**$_0$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ simulates honest parties in the whole protocol EXTEND-MULT. Note that the only difference is that, in **Hybrid**$_1$, $\mathcal{S}$ uses the real shares of $[e]_{2t}$ of honest parties, while in **Hybrid**$_2$, $\mathcal{S}$ generates random elements as the shares of $[e]_{2t}$ of honest parties. However, as we have argued in Lemma 6.1, the shares of $[r]_{2t}$ held by honest parties are uniformly random. Therefore, the shares of $[e]_{2r} := \sum_{j=1}^{\ell} [x_j]_t \cdot [y_j]_t + [r]_{2t}$ held by honest parties are also uniformly random. Thus, the distribution of **Hybrid**$_2$ is the same as **Hybrid**$_1$.

Note that **Hybrid**$_2$ is the execution in the ideal world, and the distribution of **Hybrid**$_2$ is identical to the distribution of **Hybrid**$_0$, the execution in the real world. □

7.2 Extension of the Batch-wise Multiplication Verification Technique

In this part, we review a natural extension to the batch-wise multiplication verification technique [10]. We first review the basic technique, which is used to check the correctness of a batch of multiplication tuples efficiently.

Overview of the Batch-wise Multiplication Verification Technique. For simplicity, suppose that we are working on a large enough finite field $\mathbb{K}$ (with $|\mathbb{K}| \geq 2^\kappa$, where κ is the security parameter). Given m multiplication tuples

$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(m)}]_t, [y^{(m)}]_t, [z^{(m)}]_t),$$

we want to check whether $x^{(i)} \cdot y^{(i)} = z^{(i)}$ for all $i \in \{1, 2, \ldots, m\}$.

The high-level idea is constructing three polynomials $f(\cdot), g(\cdot), h(\cdot)$ such that

$$\forall i \in [m], f(i) = x^{(i)}, g(i) = y^{(i)}, h(i) = z^{(i)}.$$

Then check whether $f \cdot g = h$. Here $f(\cdot), g(\cdot)$ are set to be degree-$(m-1)$ polynomials in $\mathbb{K}$ so that they can be determined by $\{x^{(i)}\}_{i=1}^m, \{y^{(i)}\}_{i=1}^m$ respectively. In this case, $h(\cdot)$ should be a degree-$2(m-1)$ polynomial which is determined by $2m-1$ values. To this end, for $i \in \{m+1, \ldots, 2m-1\}$, we need to compute $z^{(i)} = f(i) \cdot g(i)$ so that $h(\cdot)$ can be determined by $\{z^{(i)}\}_{i=1}^{2m-1}$.

In more detail, all parties first locally compute $[f(\cdot)]_t, [g(\cdot)]_t$ using $\{[x^{(i)}]_t\}_{i=1}^m$ and $\{[y^{(i)}]_t\}_{i=1}^m$ respectively. For $i \in \{m+1, \ldots, 2m-1\}$, all parties locally compute $[f(i)]_t, [g(i)]_t$ and then invoke $\mathcal{F}_{\text{mult-mal}}$ to compute $[z^{(i)}]_t$. Finally, all parties locally compute $[h(\cdot)]_t$ using $\{[z^{(i)}]_t\}_{i=1}^{2m-1}$.

Note that if $x^{(i)} \cdot y^{(i)} = z^{(i)}$ for all $i \in \{1, 2, \ldots, 2m-1\}$, then we have $f \cdot g = h$. Otherwise, we must have $f \cdot g \neq h$. Therefore, it is sufficient to check whether $f \cdot g = h$. Since $h(\cdot)$ is a degree-$2(m-1)$ polynomials, in the case that $f \cdot g \neq h$, the number of $x \in \mathbb{K}$ such that $f(x) \cdot g(x) = h(x)$ holds is at most $2(m-1)$. Therefore, by randomly selecting $x \in \mathbb{K}$, with probability $2(m-1)/|\mathbb{K}|$ we have $f(x) \cdot g(x) \neq h(x)$.

Therefore, to check whether $f \cdot g = h$, all parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random challenge r. All parties locally compute $[f(r)]_t, [g(r)]_t$ and $[h(r)]_t$. It is sufficient to only check whether $([f(r)]_t, [g(r)]_t, [h(r)]_t)$ is a correct multiplication tuple.

Checking the Single Multiplication Tuple. In [10], this check is done using an "expensive" MPC protocol. Since we only need to check the final multiplication tuple, the cost of this check does not affect the overall communication complexity. In [60], a random multiplication tuple is included when using the batch-wise multiplication verification technique (so that the technique applies on $m+1$ multiplication tuples). In this way, revealing the whole sharings $([f(r)]_t, [g(r)]_t, [h(r)]_t)$ does not compromise the security of the original m multiplication tuples. Therefore, all parties simply send their shares of $[f(r)]_t, [g(r)]_t, [h(r)]_t$ to all other parties and then check whether $f(r) \cdot g(r) = h(r)$.

Description of COMPRESS. In essence, this technique compresses m checks of multiplication tuples into 1 check of a single tuple. The protocol takes m multiplication tuples as input and outputs a single tuple. We refer to this protocol as COMPRESS. The description of COMPRESS appears in Protocol 7.5. The communication complexity of COMPRESS is $O(mn + n^2)$ field elements.

Figure 7.5: Protocol COMPRESS

1. The multiplication tuples are denoted by

$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(m)}]_t.[y^{(m)}]_t, [z^{(m)}]_t).$$

2. Let $f(\cdot), g(\cdot)$ be degree-$(m-1)$ polynomials such that

$$\forall i \in \{1, 2, \ldots, m\}, f(i) = x^{(i)}, g(i) = y^{(i)}.$$

   All parties locally compute $[f(\cdot)]_t$ and $[g(\cdot)]_t$ by using $\{[x^{(i)}]_t\}_{i=1}^m$ and $\{[y^{(i)}]_t\}_{i=1}^m$ respectively.
3. For all $i \in \{m+1, \ldots, 2m-1\}$, all parties locally compute $[f(i)]_t$ and $[g(i)]_t$, and then invoke $\mathcal{F}_{\text{mult-mal}}$ on $([f(i)]_t, [g(i)]_t)$ to compute $[z^{(i)}]_t = [f(i) \cdot g(i)]_t$.
4. Let $h(\cdot)$ be a degree-$2(m-1)$ polynomials such that

$$\forall i \in \{1, 2, \ldots, 2m-1\}, h(i) = z^{(i)}.$$

   All parties locally compute $[h(\cdot)]_t$ by using $\{[z^{(i)}]_t\}_{i=1}^{2m-1}$.
5. All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random field element r. If $r \in \{1, 2, \ldots, m\}$, all parties abort. Otherwise, output $([f(r)]_t, [g(r)]_t, [h(r)]_t)$.

Lemma 7.3. *If at least one multiplication tuple is incorrect, then the resulting tuple output by* COMPRESS *is incorrect with probability* $1 - \frac{3m-2}{|\mathbb{K}|}$.

Proof. Suppose there is at least one incorrect multiplication tuple. We first count the number of r which causes either an abort of the protocol or a correct output tuple.

In COMPRESS, all parties abort if $r \in \{1, 2, \ldots, m\}$. Therefore, there are m choices of r which causes an abort of the protocol. Since there is at least one incorrect multiplication tuple, we have $f \cdot g \neq h$. Since the polynomial $h - f \cdot g$ is a degree-$2(m-1)$ non-zero polynomial, the number of $r \in \mathbb{K}$ such that $h(r) - f(r) \cdot g(r) = 0$ is at most $2(m-1)$. Therefore, there are at most $2(m-1)$ choices of r which causes a correct output tuple. In total, the number of r which causes either an abort of the protocol or a correct output tuple is bounded by $m + 2(m-1) = 3m - 2$.

Since r is randomly sampled by $\mathcal{F}_{\text{coin}}$, the probability that $\mathcal{F}_{\text{coin}}$ outputs such a bad r is bounded by $\frac{3m-2}{|\mathbb{K}|}$. Therefore, with probability $1 - \frac{3m-2}{|\mathbb{K}|}$, the tuple output by COMPRESS is incorrect. □

Extension. A natural extension of the batch-wise multiplication verification technique is to check the correctness of m inner-product tuples. This idea has been observed in [60].

Given m inner-product tuples

$$\{(([x_1^{(i)}]_t, \ldots, [x_\ell^{(i)}]_t), ([y_1^{(i)}]_t, \ldots, [y_\ell^{(i)}]_t), [z^{(i)}]_t)\}_{i=1}^m,$$

we want to check whether $\sum_{j=1}^{\ell} x_j^{(i)} \cdot y_j^{(i)} = z^{(i)}$ for all $i \in \{1, 2, \ldots, m\}$. The idea is to construct two vectors of degree-$(m-1)$ polynomials $(f_1(\cdot), \ldots, f_\ell(\cdot))$ and $(g_1(\cdot), \ldots, g_\ell(\cdot))$ such that

$$\forall i \in \{1, 2, \ldots, m\} \text{ and } j \in \{1, 2, \ldots, \ell\}, f_j(i) = x_j^{(i)}, g_j(i) = y_j^{(i)}.$$

All parties can locally compute $([f_1(\cdot)]_t, \ldots, [f_\ell(\cdot)]_t)$ by using $\{([x_1^{(i)}]_t, \ldots, [x_\ell^{(i)}]_t)\}_{i=1}^m$. Similarly, they can locally compute $([g_1(\cdot)]_t, \ldots, [g_\ell(\cdot)]_t)$ by using $\{([y_1^{(i)}]_t, \ldots, [y_\ell^{(i)}]_t)\}_{i=1}^m$.

For $i \in \{m+1, \ldots, 2m-1\}$, all parties compute $([f_1(i)]_t, \ldots, [f_\ell(i)]_t)$ and $([g_1(i)]_t, \ldots, [g_\ell(i)]_t)$, and then compute the degree-t Shamir sharing $[z^{(i)}]_t$ by invoking $\mathcal{F}_{\text{extendMult}}$ on these two vectors of sharings. Let $h(\cdot)$ be a degree-$2(m-1)$ polynomial such that

$$\forall i \in \{1, 2, \ldots, 2m-1\}, h(i) = z^{(i)}.$$

All parties can locally compute $[h(\cdot)]_t$ by using $\{[z^{(i)}]_t\}_{i=1}^{2m-1}$.

The remaining steps are similar to that in COMPRESS. We refer to this extension as EXTEND-COMPRESSION. The description of EXTEND-COMPRESS appears in Protocol 7.6. The communication complexity of EXTEND-COMPRESS is $O(mn + n^2)$ field elements.

Lemma 7.4. *If at least one inner-product tuple is incorrect, then the resulting tuple output by* EXTEND-COMPRESS *is incorrect with probability* $1 - \frac{3m-2}{|\mathbb{K}|}$.

This lemma can be proved in the same way as that for Lemma 7.3. Therefore, for simplicity, we omit the details.

7.3 Our Verification Protocol

7.3.1 Step One: De-Linearization

The first step is to transform the check of m multiplication tuples into one check of an inner-product tuple of dimension m. The description of DE-LINEARIZATION appears in Protocol 7.7. The communication complexity of DE-LINEARIZATION is $O(n^2)$ elements in $\mathbb{K}$.

Lemma 7.5. *If at least one multiplication tuple is incorrect, then the resulting inner-product tuple output by* DE-LINEARIZATION *is also incorrect with overwhelming probability.*

Proof. Suppose there is at least one incorrect multiplication tuple. We first count the number of r which causes a correct output tuple.

Consider the following two polynomials of degree-$(m-1)$ in $\mathbb{K}$:

$$\begin{aligned} F(X) &= (x^{(1)} \cdot y^{(1)}) + (x^{(2)} \cdot y^{(2)})X + \ldots + (x^{(m)} \cdot y^{(m)})X^{m-1} \\ G(X) &= z^{(1)} + z^{(2)}X + \ldots + z^{(m)}X^{m-1}. \end{aligned}$$

Figure 7.6: Protocol EXTEND-COMPRESS

1. The inner-product tuples are denoted by

$$\{(([x_1^{(i)}]_t, \ldots, [x_\ell^{(i)}]_t), ([y_1^{(i)}]_t, \ldots, [y_\ell^{(i)}]_t), [z^{(i)}]_t)\}_{i=1}^m$$

2. Let $(f_1(\cdot), \ldots, f_\ell(\cdot))$ and $(g_1(\cdot), \ldots, g_\ell(\cdot))$ be vectors of degree-$(m-1)$ polynomials such that

$$\forall i \in \{1, 2, \ldots, m\} \text{ and } j \in \{1, 2, \ldots, \ell\}, f_j(i) = x_j^{(i)}, g_j(i) = y_j^{(i)}.$$

   All parties locally compute $([f_1(\cdot)]_t, \ldots, [f_\ell(\cdot)]_t)$ and $([g_1(\cdot)]_t, \ldots, [g_\ell(\cdot)]_t)$ by using $\{([x_1^{(i)}]_t, \ldots, [x_\ell^{(i)}]_t)\}_{i=1}^m$ and $\{([y_1^{(i)}]_t, \ldots, [y_\ell^{(i)}]_t)\}_{i=1}^m$ respectively.

3. For all $i \in \{m+1, \ldots, 2m-1\}$, all parties locally compute $([f_1(i)]_t, \ldots, [f_\ell(i)]_t)$ and $([g_1(i)]_t, \ldots, [g_\ell(i)]_t)$, and then invoke $\mathcal{F}_{\text{extendMult}}$ on these two vectors of sharings to compute $[z^{(i)}]_t = [\sum_{j=1}^\ell f_j(i) \cdot g_j(i)]_t$.

4. Let $h(\cdot)$ be a degree-$2(m-1)$ polynomials such that

$$\forall i \in \{1, 2, \ldots, 2m-1\}, h(i) = z^{(i)}.$$

   All parties locally compute $[h(\cdot)]_t$ by using $\{[z^{(i)}]_t\}_{i=1}^{2m-1}$.

5. All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random field element r. If $r \in \{1, 2, \ldots, m\}$, all parties abort. Otherwise, output $(([f_1(r)]_t, \ldots, [f_\ell(r)]_t), ([g_1(r)]_t, \ldots, [g_\ell(r)]_t), [h(r)]_t)$.

In the case that there exists an incorrect multiplication tuple, we have $F(\cdot) \neq G(\cdot)$. Since the polynomial $F(\cdot) - G(\cdot)$ is a degree-$(m-1)$ non-zero polynomial, the number of $r \in \mathbb{K}$ such that $F(r) - G(r) = 0$ is at most $m-1$. Therefore there are at most $m-1$ choices of r which causes a correct output tuple.

Since r is randomly sampled by $\mathcal{F}_{\text{coin}}$, the probability that $\mathcal{F}_{\text{coin}}$ outputs such a bad r is bounded by $\frac{m-1}{|\mathbb{K}|}$. Recall that κ is the security parameter and the size of $\mathbb{K}$ is at least 2^κ. Therefore, with probability $1 - \frac{m-1}{|\mathbb{K}|} \geq 1 - \frac{m-1}{2^\kappa}$, the tuple output by DE-LINEARIZATION is incorrect. □

7.3.2 Step Two: Dimension-Reduction

The second step is to reduce the dimension of the inner-product tuple output by DE-LINEARIZATION. We will use EXTEND-COMPRESS as a building block. The description of DIMENSION-REDUCTION appears in Protocol 7.8. The communication complexity of DIMENSION-REDUCTION is $O(kn + n^2)$ elements in $\mathbb{K}$, where k is the compression parameter.

Lemma 7.6. *If the input inner-product tuple is incorrect, then the resulting inner-product tuple output by* DIMENSION-REDUCTION *is also incorrect with overwhelming probability.*

Figure 7.7: Protocol DE-LINEARIZATION

1. The multiplication tuples are denoted by

$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(m)}]_t.[y^{(m)}]_t, [z^{(m)}]_t).$$

2. All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random field element $r \in \mathbb{K}$.
3. For all $j \in \{1, 2, \ldots, m\}$, all parties set $[a_j]_t = r^{j-1} \cdot [x^{(j)}]_t$ and $[b_j]_t = [y^{(j)}]_t$. All parties compute

$$[c]_t = \sum_{j=1}^{m} r^{j-1} [z^{(j)}]_t$$

and output the inner-product tuple

$$(([a_1]_t, \ldots, [a_m]_t), ([b_1]_t, \ldots, [b_m]_t), [c]_t).$$

Figure 7.8: Protocol DIMENSION-REDUCTION

1. Suppose the input inner-product tuple is denoted by

$$(([x_1]_t, \ldots, [x_m]_t), ([y_1]_t, \ldots, [y_m]_t), [z]_t).$$

Let k denote the compression parameter and $\ell = m/k$.

2. For all $i \in \{1, 2, \ldots, k\}$ and $j \in \{1, 2, \ldots, \ell\}$, all parties set $[a_j^{(i)}]_t = [x_{(i-1)\cdot\ell+j}]_t$ and $[b_j^{(i)}]_t = [y_{(i-1)\cdot\ell+j}]_t$.
3. For all $i \in \{1, 2, \ldots, k-1\}$, all parties invoke $\mathcal{F}_{\text{extendMult}}$ on $([a_1^{(i)}]_t, \ldots, [a_\ell^{(i)}]_t)$ and $([b_1^{(i)}]_t, \ldots, [b_\ell^{(i)}]_t)$ to compute $[c^{(i)}]_t = [\sum_{j=1}^{\ell} a_j^{(i)} \cdot b_j^{(i)}]_t$.
4. All parties set $[c^{(k)}]_t = [z]_t - \sum_{i=1}^{k-1} [c^{(i)}]_t$.
5. All parties invoke EXTEND-COMPRESS on the following k inner-product tuples:

$$\{(([a_1^{(i)}]_t, \ldots, [a_\ell^{(i)}]_t), ([b_1^{(i)}]_t, \ldots, [b_\ell^{(i)}]_t), [c^{(i)}]_t)\}_{i=1}^{k}.$$

The output is denoted by

$$(([a_1]_t, \ldots, [a_\ell]_t), ([b_1]_t, \ldots, [b_\ell]_t), [c]_t).$$

All parties take this new inner-product tuple as output.

Proof. Suppose that the input inner-product tuple $(([x_1]_t, \ldots, [x_m]_t), ([y_1]_t, \ldots, [y_m]_t), [z]_t)$ is incorrect. We first show that at least one of the following k inner-product tuples is incorrect:

$$\{(([a_1^{(i)}]_t, \ldots, [a_\ell^{(i)}]_t), ([b_1^{(i)}]_t, \ldots, [b_\ell^{(i)}]_t), [c^{(i)}]_t)\}_{i=1}^{k}$$

Note that the first $k-1$ tuples are computed via $\mathcal{F}_{\text{extendMult}}$. If at least one of the inner-product tuples in the first $k-1$ tuples is incorrect, then the statement holds. Assume that the first $k-1$ tuples are all correct, i.e., for all $i \in \{1, 2, \ldots, k-1\}$, $\sum_{j=1}^{\ell} a_j^{(i)} \cdot b_j^{(i)} = c^{(i)}$. Since the input inner-product tuple is incorrect, we have $\sum_{j=1}^{m} x_j \cdot y_j \neq z$. Therefore

$$\sum_{j=1}^{\ell} a_j^{(k)} \cdot b_j^{(k)} = \sum_{j=1}^{m} x_j \cdot y_j - \sum_{i=1}^{k-1} \left(\sum_{j=1}^{\ell} a_j^{(i)} \cdot b_j^{(i)} = c^{(i)} \right) \neq z - \sum_{i=1}^{k-1} c^{(i)} = c^{(k)},$$

which means that the last inner-product tuple must be incorrect.

According to Lemma 7.4, the resulting tuple $(([a_1]_t, \ldots, [a_\ell]_t), ([b_1]_t, \ldots, [b_\ell]_t), [c]_t)$ output by EXTEND-COMPRESS is incorrect with probability $1 - \frac{3k-2}{|\mathbb{K}|} \geq 1 - \frac{3k-2}{2^\kappa}$. □

7.3.3 Step Three: Randomization

In the final step, we add a random multiplication tuple when we use COMPRESS so that the verification of the resulting multiplication tuple can be done by simply opening all the sharings. The description of RANDOMIZATION appears in Protocol 7.9. The communication complexity of RANDOMIZATION is $O(mn + n^2)$ elements in $\mathbb{K}$, where m is the dimension of the inner-product tuple.

Lemma 7.7. *If the input inner-product tuple is incorrect, then at least one honest party will abort with overwhelming probability.*

Proof. Suppose that the input inner-product tuple $(([x_1]_t, \ldots, [x_m]_t), ([y_1]_t, \ldots, [y_m]_t), [z]_t)$ is incorrect. We first show that at least one of the following m multiplication tuples is incorrect:

$$([x_1]_t, [y_1]_t, [z_1]_t), \ldots, ([x_m]_t, [y_m]_t, [z_m]_t)$$

Note that the first $m-1$ tuples are computed via $\mathcal{F}_{\text{mult-mal}}$. If at least one of the multiplication tuples in the first $m-1$ tuples is incorrect, then the statement holds. Assume that the first $m-1$ tuples are all correct, i.e., for all $i \in \{1, 2, \ldots, m-1\}$, $x_i \cdot y_i = z_i$. Since the input inner-product tuple is incorrect, we have $\sum_{i=1}^{m} x_i \cdot y_i \neq z$. Therefore

$$x_m \cdot y_m = \sum_{i=1}^{m} x_i \cdot y_i - \sum_{i=1}^{m-1} x_i \cdot y_i \neq z - \sum_{i=1}^{m-1} z_i = z_m,$$

which means that the last multiplication tuple must be incorrect.

According to Lemma 7.3, the resulting tuple $([a]_t, [b]_t, [c]_t)$ output by COMPRESS is incorrect with probability $1 - \frac{3m+1}{|\mathbb{K}|} \geq 1 - \frac{3m+1}{2^\kappa}$. Note that an incorrect tuple will all honest parties aborting the protocol. □

Figure 7.9: Protocol RANDOMIZATION

1. Suppose the input inner-product tuple is denoted by
$$(([x_1]_t, \ldots, [x_m]_t), ([y_1]_t, \ldots, [y_m]_t), [z]_t).$$
2. All parties invoke two times of $\mathcal{F}_{\text{rand}}$ to prepare two random degree-t Shamir sharings $[x_0]_t, [y_0]_t$.
3. All parties invoke $\mathcal{F}_{\text{mult-mal}}$ on $([x_0]_t, [y_0]_t)$ to compute $[z_0]_t = [x_0 \cdot y_0]_t$.
4. For $i \in \{1, 2, \ldots, m-1\}$, all parties invoke $\mathcal{F}_{\text{mult-mal}}$ on $([x_i]_t, [y_i]_t)$ to compute $[z_i]_t = [x_i \cdot y_i]_t$. Then set
$$[z_m]_t = [z]_t - \sum_{i=1}^{m-1} [z_i]_t.$$
5. All parties invoke COMPRESS on
$$([x_0]_t, [y_0]_t, [z_0]_t), ([x_1]_t, [y_1]_t, [z_1]_t), \ldots, ([x_m]_t, [y_m]_t, [z_m]_t).$$
The output is denoted by $([a]_t, [b]_t, [c]_t)$.
6. All parties send their shares of $[a]_t, [b]_t, [c]_t$ to all other parties.
7. All parties reconstruct a, b, c. For each party P_i, if the shares of $[a]_t, [b]_t, [c]_t$ are inconsistent or $a \cdot b \neq c$, P_i aborts. Otherwise, P_i takes `accept` as output.

7.3.4 Summary

In this section, we show how to check the correctness of m multiplication tuples with communication complexity $o(m)$. It is a simple combination of the three protocols DE-LINEARIZATION, DIMENSION-REDUCTION, and RANDOMIZATION. The description of MULTVERIFICATION appears in Protocol 7.10.

Lemma 7.8. *The protocol* MULTVERIFICATION *securely computes $\mathcal{F}_{\text{multVerify}}$ with abort in the $\{\mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{mult-mal}}, \mathcal{F}_{\text{extendMult}}\}$-hybrid model in the presence of a fully malicious adversary controlling t corrupted parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Recall that $\mathcal{C}orr$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation of MULTVERIFICATION. In the beginning, $\mathcal{S}$ receives from $\mathcal{F}_{\text{multVerify}}$ the shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$ held by corrupted parties and the difference $d^{(i)} = z^{(i)} - x^{(i)} \cdot y^{(i)}$ for all $i \in \{1, 2, \ldots, m\}$. Furthermore, $\mathcal{S}$ receives $b \in \{\texttt{abort}, \texttt{accept}\}$ indicating whether the multiplications are correct.

Figure 7.10: Protocol MULTVERIFICATION

1. Let k be the compression parameter, m denote the number of multiplication tuples. The multiplication tuples are denoted by

$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(m)}]_t.[y^{(m)}]_t, [z^{(m)}]_t).$$

2. All parties invoke DE-LINEARIZATION on these m multiplication tuples. Let

$$(([a_1]_t, \ldots, [a_m]_t), ([b_1]_t, \ldots, [b_m]_t), [c]_t)$$

   denote the output.

3. Let ℓ denote the dimension of the inner-product tuple. Initially $\ell = m$. While $\ell \geq k$, all parties run the following steps.
   (a) All parties invoke DIMENSION-REDUCTION on the current inner-product tuple and obtain a new inner-product tuple, denoted by

$$(([a_1]_t, \ldots, [a_{\ell/k}]_t), ([b_1]_t, \ldots, [b_{\ell/k}]_t), [c]_t).$$

   (b) All parties set $\ell := \ell/k$.

4. Let

$$(([a_1]_t, \ldots, [a_k]_t), ([b_1]_t, \ldots, [b_k]_t), [c]_t)$$

   denote the inner-product tuple from the last step. All parties invoke RANDOMIZATION on this inner-product tuple.

- Simulation of DE-LINEARIZATION:
  When $\mathcal{F}_{\text{coin}}$ is invoked in Step 2, $\mathcal{S}$ emulates $\mathcal{F}_{\text{coin}}$ and generates a random field element $r \in \mathbb{K}$. Then, $\mathcal{S}$ computes the shares of $(([a_1]_t, \ldots, [a_m]_t), ([b_1]_t, \ldots, [b_m]_t), [c]_t)$ held by corrupted parties and the difference $d = c - \sum_{j=1}^{m} a_j \cdot b_j$. This can be achieved by using the shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$ held by corrupted parties and the difference $d^{(i)} = z^{(i)} - x^{(i)} \cdot y^{(i)}$ for all $i \in \{1, 2, \ldots, m\}$.
- Simulation of DIMENSION-REDUCTION:
  We will maintain the invariance that, for the input inner-product tuple

$$(([x_1]_t, \ldots, [x_m]_t), ([y_1]_t, \ldots, [y_m]_t), [z]_t),$$

  $\mathcal{S}$ learns the shares held by corrupted parties and the difference $d = z - \sum_{j=1}^{m} x_j \cdot y_j$. Note that this is true for the first time of invocation of DIMENSION-REDUCTION since these shares and the difference are computed when simulating DE-LINEARIZATION.
  In Step 2 and Step 3, $\mathcal{S}$ computes the shares of $(([a_1^{(i)}]_t, \ldots, [a_\ell^{(i)}]_t), ([b_1^{(i)}]_t, \ldots, [b_\ell^{(i)}]_t), [c^{(i)}]_t)$ held by corrupted parties and the difference $d^{(i)} = c^{(i)} - \sum_{j=1}^{\ell} a_j^{(i)} \cdot b_j^{(i)}$ for all $i \in \{1, 2, \ldots, k\}$. Concretely,

- For all $i \in \{1, 2, \ldots, k\}$, the shares of $([a_1^{(i)}]_t, \ldots, [a_\ell^{(i)}]_t)$ and $([b_1^{(i)}]_t, \ldots, [b_\ell^{(i)}]_t)$ held by corrupted parties can be directly obtained from the shares of $([x_1]_t, \ldots, [x_m]_t)$ and $([y_1]_t, \ldots, [y_m]_t)$ held by corrupted parties.
- For all $i \in \{1, 2, \ldots, k-1\}$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{extendMult}}$ and receives from the adversary the shares of $[c^{(i)}]_t$ held by corrupted parties. $\mathcal{S}$ also receives the difference $d^{(i)}$ from the adversary.
- For $[c^{(k)}]_t$ and $d^{(k)}$, recall that $[c^{(k)}]_t$ is computed by

$$[c^{(k)}]_t = [z]_t - \sum_{i=1}^{k-1} [c^{(i)}]_t.$$

  Therefore, $\mathcal{S}$ can computes the shares held by corrupted parties from the above equation. Furthermore, $\mathcal{S}$ computes $d^{(k)}$ by

$$d^{(k)} = d - \sum_{i=1}^{k-1} d^{(i)}.$$

In Step 4, $\mathcal{S}$ needs to simulate the behaviors of honest parties in EXTEND-COMPRESS. Note that EXTEND-COMPRESS only contains local computation and invocations of $\mathcal{F}_{\text{extendMult}}$ and $\mathcal{F}_{\text{coin}}$. For $\mathcal{F}_{\text{coin}}$, $\mathcal{S}$ faithfully generates a random element. For $\mathcal{F}_{\text{extendMult}}$, $\mathcal{S}$ receives from the adversary the shares of corrupted parties and the difference. $\mathcal{S}$ follows EXTEND-COMPRESS to computes the shares of $(([a_1]_t, \ldots, [a_\ell]_t), ([b_1]_t, \ldots, [b_\ell]_t), [c]_t)$ held by corrupted parties and the difference $d' = c - \sum_{j=1}^{\ell} a_j \cdot b_j$.

- Simulation of RANDOMIZATION:
  Note that, for the input inner-product tuple $(([x_1]_t, \ldots, [x_m]_t), ([y_1]_t, \ldots, [y_m]_t), [z]_t)$, the simulator $\mathcal{S}$ learns the shares held by corrupted parties and the difference $d = z - \sum_{i=1}^{m} x_i \cdot y_i$ since they are computed when simulating DIMENSION-REDUCTION.
  In RANDOMIZATION, for all $i \in \{1, 2, \ldots, m\}$, $\mathcal{S}$ computes the shares of $[x_i]_t, [y_i]_t, [z_i]_t$ held by corrupted parties and the difference $d_i = z_i - x_i \cdot y_i$ in the same way as that in DIMENSION-REDUCTION. For $[x_0]_t, [y_0]_t$, $\mathcal{S}$ receives from the adversary the shares held by corrupted parties when emulating $\mathcal{F}_{\text{rand}}$. For $[z_0]_t$, $\mathcal{S}$ receives the shares from the adversary held by corrupted parties and the difference d_0 when emulating $\mathcal{F}_{\text{mult-mal}}$.
  In Step 6, $\mathcal{S}$ needs to simulate the behaviors of honest parties in COMPRESS. This can be simulated in the same way as that for EXTEND-COMPRESS. At the end of this step, $\mathcal{S}$ learns the shares of $[a]_t, [b]_t, [c]_t$ held by corrupted parties and also the difference $d' = c - a \cdot b$. $\mathcal{S}$ randomly samples a, b and computes $c = a \cdot b + d'$. Based on the secrets a, b, c and the shares held by corrupted parties, $\mathcal{S}$ reconstructs the whole sharings $[a]_t, [b]_t, [c]_t$.
  In Step 7 and Step 8, $\mathcal{S}$ faithfully follows the protocol. Recall that $\mathcal{S}$ receives $b \in \{\texttt{abort}, \texttt{accept}\}$ from $\mathcal{F}_{\text{multVerify}}$ indicating whether the multiplications are correct. If $b = \texttt{accept}$ but an honest party takes `abort` as output, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{multVerify}}$. Otherwise, $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\text{multVerify}}$.

Hybrids Argument. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties with overwhelming probability. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ computes the difference for each multiplication tuple and inner-product tuple as described above. Note that this does not change the behaviors of honest parties. Therefore, the distribution of **Hybrid**$_0$ is identical to **Hybrid**$_1$.

Hybrid$_2$: In this hybrid, instead of using the real sharings $[a]_t, [b]_t, [c]_t$ in RANDOMIZATION, $\mathcal{S}$ constructs the sharings $[a]_t, [b]_t, [c]_t$ as described above. Concretely, $\mathcal{S}$ randomly samples the secrets a, b and reconstructs the whole sharings $[a]_t, [b]_t$ based on the shares held by corrupted parties. Then based on the difference d' of this tuple, $\mathcal{S}$ computes $c = a \cdot b + d'$ and reconstructs the whole sharing $[c]_t$ based on the shares held by corrupted parties.

Note that in RANDOMIZATION, a and b are linear combinations of $\{x_i\}_{i=0}^m$ and $\{y_i\}_{i=0}^m$ respectively and the coefficients are all non-zero, where the latter follows from the property of polynomials. Also note that x_0 and y_0 are randomly chosen by $\mathcal{F}_{\text{rand}}$. Thus, a and b are uniformly random. The only difference between **Hybrid**$_1$ and **Hybrid**$_2$ is that, in **Hybrid**$_1$, a and b are masked by x_0 and y_0 which are randomly chosen by $\mathcal{F}_{\text{rand}}$, while in **Hybrid**$_2$, a and b are randomly chosen by $\mathcal{S}$. However the distributions of a and b remains unchanged. Since c is determined by a, b and the difference d', the distribution of c remains the same in both hybrids.

Therefore, the distribution of **Hybrid**$_2$ is identical to **Hybrid**$_1$.

Hybrid$_3$: In this hybrid, if at least one of the input multiplication tuples is incorrect, $\mathcal{S}$ aborts in the end of the protocol. Note that in the real protocol, it is possible that while one of the input multiplication tuples is incorrect, the final multiplication tuple verified in RANDOMIZATION is correct. In this case, honest parties in **Hybrid**$_2$ do not abort.

However, according to Lemma 7.5, Lemma 7.6, Lemma 7.7, this happens with negligible probability.

Therefore, the distribution of **Hybrid**$_3$ is statistically close to **Hybrid**$_2$.

Hybrid$_4$: In this hybrid, $\mathcal{S}$ simulates the behaviors of honest parties as described above. Note that the only place where honest parties need to communicate with corrupted parties is Step 7 in RANDOMIZATION where all parties verify the correctness of the final multiplication tuple. However, the preparation of $[a]_t, [b]_t, [c]_t$ only depends on the shares and the differences of the input multiplication tuples, which can be obtained from $\mathcal{F}_{\text{multVerify}}$.

Therefore, the distribution of **Hybrid**$_4$ is identical to **Hybrid**$_3$.

Note that **Hybrid**$_4$ is the execution in the ideal world, and the distribution of **Hybrid**$_4$ is statistically close to the distribution of **Hybrid**$_0$, the execution in the real world. □

Concrete Efficiency. Now we analyze the communication complexity of MULTVERIFICATION. Recall that each time of running DIMENSION-REDUCTION reduces the dimension of the inner-product tuple to be $1/k$ of the original dimension. Therefore, MULTVERIFICATION includes 1 invocation of DE-LINEARIZATION, $(\log_k m - 1)$ invocations of DIMENSION-REDUCTION and 1 invocation of RANDOMIZATION. The communication complexity of MULTVERIFICATION is

$$O(n^2) + (\log_k m - 1) \cdot O(kn + n^2) + O(kn + n^2) = O((kn + n^2) \log_k m)$$

field elements in $\mathbb{K}$.

Remark 7.1. *We note that the circuit size is bounded by poly(κ) where κ is the security parameter. Therefore, if we set $k = \kappa$, the communication complexity of* MULTVERIFICATION *becomes*

$O(n\kappa + n^2)$ *field elements in* $\mathbb{K}$.

Remark 7.2. *Note that* MULTVERIFICATION *requires* $O(\log_k m)$ *rounds. In the real world, one can adjust* k *based on the overhead of each round and the overhead of sending each bit via a private channel to achieve the best running time.*

7.4 Compiling the Semi-Honest DN Protocol with Malicious Security

For completeness, we show how to use our fast verification protocol to compile the semi-honest DN protocol [28] to a maliciously secure protocol without affecting the concrete efficiency. Recall that we are in the client-server model where there are c clients and $n = 2t + 1$ servers (denoted by parties). The adversary is able to control up to c clients and t parties. The clients only participate in the input phase and the output phase.

At a high-level, the main protocol is consist of the following steps:

1. Input: Each client uses degree-t Shamir secret sharing scheme to share its inputs to the parties.
2. Evaluation Phase: All parties evaluate the circuit gate by gate.
3. Verification Phase: All parties check the correctness of multiplications.
4. Output: For each output gate, all parties send their shares to the client who should receive it.

The functionality $\mathcal{F}_{\text{main-mal}}$ is described in Functionality 7.11. The main protocol DN-MAIN-MAL appears in Protocol 7.12. The communication complexity of DN-MAIN-MAL is $O(|C| \cdot n + \log_k |C| \cdot (kn + n^2)\kappa)$ field elements, where $|C|$ is the circuit size, k is the compression parameter, and κ is the security parameter.

Figure 7.11: Functionality $\mathcal{F}_{\text{main-mal}}$

1. $\mathcal{F}_{\text{main-mal}}$ receives from all clients their inputs.
2. $\mathcal{F}_{\text{main-mal}}$ evaluates the circuits and computes the output. $\mathcal{F}_{\text{main-mal}}$ first sends the output of corrupted clients to the adversary.
   - If the adversary replies `continue`, $\mathcal{F}_{\text{main-mal}}$ distributes the output to honest clients.
   - If the adversary replies `abort`, $\mathcal{F}_{\text{main-mal}}$ sends `abort` to honest clients.

Lemma 7.9. *Let* c *be the number of clients and* $n = 2t + 1$ *be the number of parties. The protocol* DN-MAIN-MAL *securely computes* $\mathcal{F}_{\textit{main-mal}}$ *with abort in the* $\{\mathcal{F}_{\textit{mult-mal}}, \mathcal{F}_{\textit{multVerify}}\}$*-hybrid model in the presence of a fully malicious adversary controlling up to* c *clients and* t *parties.*

Figure 7.12: Protocol DN-MAIN-MAL

1. **Input Phase**:
   Let $\mathsf{Client}_1, \ldots, \mathsf{Client}_c$ denote the clients who provide inputs. For every input gate of Client_i, Client_i samples a random degree-t Shamir sharing of its input, $[x]_t$, and distributes the shares to all parties.
2. **Evaluation Phase**:
   All parties evaluate the circuit layer by layer.
   - For each addition gate with input sharings $[x]_t, [y]_t$, all parties locally compute the output sharing $[z]_t := [x]_t + [y]_t$.
   - For each multiplication gate with input sharings $[x]_t, [y]_t$, all parties invoke $\mathcal{F}_{\text{mult-mal}}$ to compute the output sharing $[z]_t = [x \cdot y]_t$.
3. **Verification Phase**:
   All parties invoke $\mathcal{F}_{\text{multVerify}}$ to check the correctness of the multiplications.
4. **Output Phase**:
   For each output gate of Client_i, let $[x]_t$ denote the input sharing. All parties send their shares of $[x]_t$ to Client_i. Client_i checks whether the shares of $[x]_t$ is consistent. If not, Client_i aborts. Otherwise, Client_i reconstructs the result x.

Proof. As we argued in Chapter 2, it is sufficient to focus on adversaries who control exactly t parties. The correctness of DN-MAIN-MAL follows from the description.

Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties and honest clients. Recall that $\mathcal{C}orr$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation of DN-MAIN-MAL. We describe the strategy of $\mathcal{S}$ phase by phase.

- Simulation of Input Phase:
  For an input x belongs to an honest client, $\mathcal{S}$ randomly samples t elements as the shares of $[x]_t$ of corrupted parties. Then $\mathcal{S}$ sends these shares to corrupted parties.
  For an input x belongs to a corrupted client, $\mathcal{S}$ receives from the adversary the shares held by honest parties. Note that, $\mathcal{S}$ learns $t+1$ shares of $[x]_t$. $\mathcal{S}$ reconstructs the whole sharing $[x]_t$ and sends x as the input of this corrupted client to $\mathcal{F}_{\text{main-mal}}$.
  Note that for each input sharing, $\mathcal{S}$ learns the shares held by corrupted parties.
- Simulation of Evaluation Phase:
  In the evaluation phase, $\mathcal{S}$ will compute the shares of each sharing held by corrupted parties. Note that this already holds for the sharing of each input gate.
  - For each addition gate with input sharings $[x]_t, [y]_t$, $\mathcal{S}$ computes the shares of $[z]_t = [x]_t + [y]_t$ held by corrupted parties.

- For each multiplication gate with input sharings $[x]_t, [y]_t$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{mult-mal}}$ and receives the shares of $[z]_t$ held by corrupted parties and the difference d.

Note that for each multiplication tuple $([x]_t, [y]_t, [z]_t)$, $\mathcal{S}$ learns the shares held by corrupted parties and the difference $d = z - x \cdot y$.

- Simulation of Verification Phase:

  $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{multVerify}}$. Recall that in the simulation of the evaluation phase, $\mathcal{S}$ has computed the shares of each multiplication tuple held by corrupted parties and the difference. $\mathcal{S}$ directly sends these shares and differences to the adversary as in $\mathcal{F}_{\text{multVerify}}$. If there exists a non-zero difference, $\mathcal{S}$ sets $b = \texttt{abort}$. Otherwise, $\mathcal{S}$ sets $b = \texttt{accept}$. Then $\mathcal{S}$ sends b to the adversary.

  - If $b = \texttt{accept}$ and the adversary replies $\texttt{continue}$, $\mathcal{S}$ moves to the next phase.
  - Otherwise, $\mathcal{S}$ sends $\texttt{abort}$ to $\mathcal{F}_{\text{main-mal}}$ and aborts.

- Simulation of Output Phase:

  For each output gate with $[x]_t$ associated with it, if the receiver is an honest client, $\mathcal{S}$ receives from the adversary the shares held by corrupted parties. Then $\mathcal{S}$ checks whether the shares are the same as the ones computed by $\mathcal{S}$. If true, $\mathcal{S}$ accepts the output. Otherwise, $\mathcal{S}$ rejects the output.

  If the receiver is a corrupted client, $\mathcal{S}$ receives the result x from $\mathcal{F}_{\text{main-mal}}$. Then, based on the shares of $[x]_t$ held by corrupted parties and the secret x, $\mathcal{S}$ reconstructs the shares held by honest parties, and sends these shares to the adversary.

  Finally, if $\mathcal{S}$ rejects any output of honest clients, $\mathcal{S}$ sends $\texttt{abort}$ to $\mathcal{F}_{\text{main-mal}}$. Otherwise, $\mathcal{S}$ sends $\texttt{continue}$ to $\mathcal{F}_{\text{main-mal}}$.

Hybrids Argument. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ computes the input of corrupted clients and send them to $\mathcal{F}_{\text{main-mal}}$. The distribution of **Hybrid**$_1$ is identical to **Hybrid**$_0$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ simulates the output phase as described above. Note that the output phase is executed only when the computation is correct. For an output gate with $[x]_t$ associated with it, if the receiver is an honest client, the shares that corrupted parties should hold are determined by the shares of honest parties. Therefore, if corrupted parties send different shares from the ones computed by $\mathcal{S}$, the shares of $[x]_t$ will be inconsistent and the client will reject the output. If the receiver is a corrupted client, the shares of honest parties are determined by the output x and the shares held by corrupted parties. Therefore, the shares prepared by $\mathcal{S}$ are identical to the real shares held by honest parties.

Therefore, the distribution of **Hybrid**$_2$ is identical to the distribution of **Hybrid**$_1$.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ computes the difference of each multiplication tuple in the evaluation phase. Then $\mathcal{S}$ simulates the verification phase. Note that $\mathcal{F}_{\text{multVerify}}$ simply checks whether there is an incorrect multiplication tuple, which is equivalent to check whether there is a non-zero difference.

Therefore, the distribution of **Hybrid**$_3$ is identical to the distribution of **Hybrid**$_2$.

Hybrid$_4$: In this hybrid, $\mathcal{S}$ simulates the evaluation phase. Note that $\mathcal{S}$ receives the shares of corrupted parties and the difference when emulating $\mathcal{F}_{\text{mult-mal}}$. These are sufficient to simulating the verification phase and the output phase. Since there is no communication in this phase, the distribution of **Hybrid**$_4$ is identical to the distribution of **Hybrid**$_3$.

Hybrid$_5$: In this hybrid, $\mathcal{S}$ simulates the input phase. The only difference between **Hybrid**$_4$ and **Hybrid**$_5$ is that, in **Hybrid**$_4$, $\mathcal{S}$ uses the real input of honest clients to generate the input sharings, while in **Hybrid**$_5$, $\mathcal{S}$ simply samples random elements as the shares of corrupted parties. Note that the distributions of the shares of corrupted parties in both hybrids are the same. Therefore, the distribution of **Hybrid**$_5$ is the same as **Hybrid**$_4$.

Note that **Hybrid**$_5$ is the execution in the ideal world, and the distribution of **Hybrid**$_5$ is identical to the distribution of **Hybrid**$_0$, the execution in the real world. □

Analysis of the Concrete Efficiency. We point out that, without MULTVERIFICATION, DN-MAIN-MAL is the same as the semi-honest DN protocol [28]. The cost per multiplication gate is 6 field elements in $\mathbb{F}$ per party, including 4 field elements to prepare a pair of random double sharings, 1 element sending to $P_{\texttt{king}}$, 1 element receiving from $P_{\texttt{king}}$. Note that the cost of MULTVERIFICATION is bounded by $O(\log_k |C| \cdot (kn + n^2)\kappa)$ bits, which does not influence the cost per multiplication gate. Therefore, DN-MAIN-MAL achieves the same concrete efficiency as the semi-honest DN protocol [28]. When setting the compression parameter $k = \kappa$, the cost of MULTVERIFICATION is bounded by $O(n^2 \cdot \kappa + n \cdot \kappa^2)$ bits. Thus, the overall communication complexity of DN-MAIN-MAL is bounded by $O(|C| \cdot n + n^2 \cdot \kappa + n \cdot \kappa^2)$ field elements.

Chapter 8

Achieving Malicious Security

In this chapter, we discuss how to compile our two semi-honest protocols constructed in Chapter 6 to maliciously secure protocols. We refer the first protocol that uses the trick of t-wise independence (See Chapter 6.2) as the `t-wise` variant, and refer the second protocol that reduces the round complexity (See Chapter 6.3) as the `round-compression` variant.

8.1 Achieving Malicious Security for the `t-wise` Variant

For the `t-wise` variant, we show that after replacing the DN multiplication protocol by our efficient multiplication protocol ATLAS-MULT (Protocol 6.7), the whole semi-honest protocol is still secure up to an additive attack. Then we can achieve malicious security as follows:

1. Run the `t-wise` variant until the output phase.
2. Check the correctness of multiplication tuples using our fast verification protocol.
3. Reconstruct the output only if the check passes.

To this end, we consider the scenario where all parties evaluate *a sequence of* N multiplication gates. In particular, the input sharings of each multiplication gate can depend on the input sharings or output sharings of the previous multiplication gates. The functionality $\mathcal{F}'_{\text{mult-mal}}$ appears in Functionality 8.1, which invokes $\mathcal{F}_{\text{mult-mal}}$ for each multiplication gate. One can view $\mathcal{F}'_{\text{mult-mal}}$ as an interface of $\mathcal{F}_{\text{mult-mal}}$. We show that the protocol ATLAS-MULT already realizes $\mathcal{F}'_{\text{mult-mal}}$ in the presence of a fully malicious adversary.

Figure 8.1: Functionality $\mathcal{F}'_{\text{mult-mal}}$

1. $\mathcal{F}'_{\text{mult-mal}}$ receives N from all parties.
2. From $i = 1$ to N, let $[x^{(i)}]_t, [y^{(i)}]_t$ denote the input sharings of the i-th multiplication gate. $\mathcal{F}'_{\text{mult-mal}}$ invokes $\mathcal{F}_{\text{mult-mal}}$ on $[x^{(i)}]_t, [y^{(i)}]_t$.

Lemma 8.1. *The protocol* ATLAS-MULT *securely computes the functionality $\mathcal{F}'_{\text{mult-mal}}$ in the $\mathcal{F}_{\text{doubleRand}}$-hybrid model in the presence of a fully malicious adversary controlling t corrupted*

parties.

Proof. Recall that $\mathcal{C}orr$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties. Recall that in Lemma 6.1, we have argued that the random degree-$2t$ sharing $[r]_{2t}$ output by $\mathcal{F}_{\text{doubleRand}}$ satisfies that the shares of honest parties are uniformly random, and are independent of the shares chosen by the adversary.

Recall that in EXPAND, all parties compute

$$\begin{aligned}([\tilde{r}^{(1)}]_t, \ldots, [\tilde{r}^{(n)}]_t)^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_t, \ldots, [r^{(t)}]_t)^{\mathrm{T}} \\ ([\tilde{r}^{(1)}]_{2t}, \ldots, [\tilde{r}^{(n)}]_{2t})^{\mathrm{T}} &= \boldsymbol{M}([r^{(1)}]_{2t}, \ldots, [r^{(t)}]_{2t})^{\mathrm{T}},\end{aligned}$$

where $\boldsymbol{M}$ is a Vandermonde matrix. From the above argument, the shares of $\{[r^{(i)}]_{2t}\}_{i=1}^{t}$ held by honest parties are uniformly random. According to the property of Vandermonde matrices, there is a one-to-one map from $\{[r^{(i)}]_{2t}\}_{i=1}^{t}$ to $\{[\tilde{r}^{(i)}]_{2t}\}_{i\in\mathcal{C}orr}$. Thus, the shares of $\{[\tilde{r}^{(i)}]_{2t}\}_{i\in\mathcal{C}orr}$ held by honest parties are uniformly random. This means that for all double sharings in the form of $([r]_t, [r]_{2t}, P_j)$ output by EXPAND, where P_j is a corrupted party, the shares of $[r]_{2t}$ held by honest parties are uniformly random.

Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties.

Simulation for ATLAS-MULT. In the first step, $\mathcal{S}$ emulates $\mathcal{F}_{\text{doubleRand}}$ and receives the shares of random double sharings held by corrupted parties. Then $\mathcal{S}$ follows EXPAND and computes the shares of the double sharings in the form of $([r]_t, [r]_{2t}, P_j)$ held by corrupted parties.

In the second step, we describe the strategy of $\mathcal{S}$ for each invocation of MULT.

1. Let $[x^{(i)}]_t, [y^{(i)}]_t$ denote the input sharings. $\mathcal{S}$ receives from $\mathcal{F}'_{\text{mult-mal}}$ the shares of $[x^{(i)}]_t, [y^{(i)}]_t$ held by corrupted parties. Let $([r]_t, [r]_{2t}, P_j)$ be the first pair of unused double sharings. Recall that $\mathcal{S}$ has learnt the shares of $[r]_t, [r]_{2t}$ held by corrupted parties.
2. $\mathcal{S}$ computes the shares of $[e^{(i)}]_{2t} := [x^{(i)}]_t \cdot [y^{(i)}]_t + [r]_{2t}$ held by corrupted parties. If P_j is corrupted, $\mathcal{S}$ samples random elements as shares of $[e^{(i)}]_{2t}$ of honest parties.
3. Depending on whether P_j is honest, there are two cases:
   - If P_j is honest, $\mathcal{S}$ receives the shares from corrupted parties. Let $[\tilde{e}^{(i)}]_{2t}$ denote the sharing when using the shares received from corrupted parties. This is to distinguish from $[e^{(i)}]_{2t}$ which uses the shares computed by $\mathcal{S}$ for corrupted parties. Then $\mathcal{S}$ computes the shares of $[d^{(i)}]_{2t} := [\tilde{e}^{(i)}]_{2t} - [e^{(i)}]_{2t}$ held by corrupted parties. Note that the shares of $[d^{(i)}]_{2t}$ held by honest parties are 0. $\mathcal{S}$ reconstructs the secret d. Finally for $[e^{(i)}]_t$, $\mathcal{S}$ samples random elements as the shares of corrupted parties.
   - If P_j is corrupted, $\mathcal{S}$ sends the shares of $[e^{(i)}]_{2t}$ of honest parties to P_j. Note that for $[e^{(i)}]_{2t}$, $\mathcal{S}$ knows all the shares. $\mathcal{S}$ reconstructs the secret e. Then $\mathcal{S}$ receives the shares of $[e^{(i)}]_t$ of honest parties from P_j. $\mathcal{S}$ reconstructs the whole sharing and computes the secret of $[e^{(i)}]_t$, denoted by $\tilde{e}^{(i)}$. Finally, $\mathcal{S}$ computes $d^{(i)} := \tilde{e}^{(i)} - e^{(i)}$.
4. In the last step, $\mathcal{S}$ computes the shares of $[z^{(i)}]_t := [e^{(i)}]_t - [\tilde{r}^{(i)}]_t$ held by corrupted parties. Then $\mathcal{S}$ sends the difference $d^{(i)}$ and the shares of $[z^{(i)}]_t$ held by corrupted parties to $\mathcal{F}'_{\text{mult-mal}}$.

Hybrid Arguments. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Consider the following hybrids.

$\mathbf{Hybrid}_0$: The execution in the real world.

$\mathbf{Hybrid}_1$: In this hybrid, $\mathcal{S}$ computes the shares of corrupted parties as described above. For each multiplication tuple $([x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t)$, $\mathcal{S}$ computes the difference $d^{(i)} := z^{(i)} - x^{(i)} \cdot y^{(i)}$ using the shares of honest parties. Then for all $i \in \{1, 2, \ldots, N\}$, $\mathcal{S}$ sends the difference $d^{(i)}$ and the shares of $[z^{(i)}]_t$ held by corrupted parties to $\mathcal{F}'_{\text{mult-mal}}$. Note that the shares of honest parties are determined by the shares of corrupted parties and the secrets. Therefore, the distribution of $\mathbf{Hybrid}_1$ is identical to the distribution of $\mathbf{Hybrid}_0$.

$\mathbf{Hybrid}_2$: In this hybrid, $\mathcal{S}$ computes the difference as described above. Note that:

- When P_j is honest, corrupted parties can change the multiplication result by sending incorrect shares of $[e^{(i)}]_{2t}$ to P_j. Therefore, the difference of this multiplication tuple is the secret of the sharing $[\tilde{e}^{(i)}]_{2t} - [e^{(i)}]_{2t}$, where $[e^{(i)}]_{2t}$ is the sharing when using the shares computed by $\mathcal{S}$ for corrupted parties, and $[\tilde{e}^{(i)}]_{2t}$ is the sharing when using the shares received from corrupted parties. Therefore, the difference $d^{(i)}$ computed by $\mathcal{S}$ is identical to that in $\mathbf{Hybrid}_1$.
- When P_j is corrupted, $\mathcal{S}$ can learn the value $\tilde{e}^{(i)}$ reconstructed by P_j from the shares of $[e^{(i)}]_t$ held by honest parties. $\mathcal{S}$ can also compute the correct $e^{(i)}$. Therefore, the difference $d^{(i)}$ computed by $\mathcal{S}$ is identical to that in $\mathbf{Hybrid}_1$.

Therefore, the distribution of $\mathbf{Hybrid}_2$ is identical to the distribution of $\mathbf{Hybrid}_1$.

$\mathbf{Hybrid}_3$: In this hybrid, $\mathcal{S}$ uses random elements as shares of $[e^{(i)}]_{2t}$ of honest parties when P_j is corrupted. Recall that we have shown that for $([r]_t, [r]_{2t}, P_j)$ where P_j is corrupted, the shares of $[r]_{2t}$ held by honest parties are uniformly random. Therefore, the shares of $[e^{(i)}]_{2t}$ are also uniformly random in this case. The distribution of $\mathbf{Hybrid}_3$ is identical to the distribution of $\mathbf{Hybrid}_2$.

$\mathbf{Hybrid}_4$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\text{doubleRand}}$ and does not generate the shares of honest parties. Note that these shares are not used in $\mathbf{Hybrid}_3$. Therefore, the distribution of $\mathbf{Hybrid}_4$ is identical to the distribution of $\mathbf{Hybrid}_3$.

Note that $\mathbf{Hybrid}_4$ is the execution in the ideal world, and the distribution of $\mathbf{Hybrid}_4$ is identical to the distribution of $\mathbf{Hybrid}_0$, the execution in the real world. □

Using $\mathcal{F}'_{\text{mult-mal}}$ in the Protocol DN-MAIN-MAL. In the Protocol DN-MAIN-MAL (Protocol 7.12), all parties invoke $\mathcal{F}_{\text{mult-mal}}$ for each multiplication gate. Note that $\mathcal{F}'_{\text{mult-mal}}$ invoke $\mathcal{F}_{\text{mult-mal}}$ for each multiplication. Therefore, we view $\mathcal{F}'_{\text{mult-mal}}$ as an interface of $\mathcal{F}_{\text{mult-mal}}$. All parties initialize $\mathcal{F}'_{\text{mult-mal}}$ in the beginning of the protocol with the number of multiplications they need to compute (which is determined by the circuit). Then we replace each invocation of $\mathcal{F}_{\text{mult-mal}}$ by $\mathcal{F}'_{\text{mult-mal}}$. Following from Lemma 7.9, we conclude the following theorem.

Theorem 1.1. *Let n be a positive integer and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n + 1$. For all arithmetic circuit C over $\mathbb{F}$, there is an information-theoretic n-party MPC protocol, which achieves malicious security (with abort) against a fully malicious adversary controlling $t < n/2$ corrupted parties, with communication complexity $O(|C| \cdot n + n^2 \cdot \kappa + n \cdot \kappa^2)$ elements, where*

κ is the security parameter and $|C|$ is the circuit size. Furthermore, the concrete efficiency is 4 elements per party per multiplication gate.

8.2 Achieving Malicious Security for the `round-compression` Variant

For the `round-compression` variant, recall that the evaluation is done by first partitioning the circuit into a sequence of two-layer sub-circuits and then evaluating each sub-circuit. For multiplication gates in the first layer of all sub-circuits, we use the improved DN multiplication protocol IMPROVED-DN-MULT to compute the outputs. For multiplication gates in the second layer of all sub-circuits, we use our efficient multiplication protocol MULT to prepare Beaver triples. In particular, the correctness requires $P_{\texttt{king}}$ to distribute the *same value* to all other parties for each multiplication gate in the first layer of all sub-circuits.

Thus, an adversary can launch attack in the following two places:

- An adversary may distribute different values to other parties when $P_{\texttt{king}}$ is corrupted in IMPROVED-DN-MULT.
- An adversary may launch additive attacks in the multiplication protocols IMPROVED-DN-MULT and MULT.

We will check these two points separately. We first show how to check that all parties receive the same values from $P_{\texttt{king}}$ in IMPROVED-DN-MULT.

Checking Consistency. The idea is to compute a random linear combination of the values they received in IMPROVED-DN-MULT and exchange their results. If a party receives different values, this party will abort. We will use the functionality $\mathcal{F}_{\text{coin}}$ introduced in Chapter 5 to generate a random element. The protocol CHECKCONSISTENCY appears in Protocol 8.2. Recall that the communication complexity of the instaniation of $\mathcal{F}_{\text{coin}}$ is $O(n^2 \cdot \kappa)$ bits. The communication complexity of CHECKCONSISTENCY is $O(n^2 \cdot \kappa)$ bits.

Figure 8.2: Protocol CHECKCONSISTENCY$(N, \{x^{(1)}, \ldots, x^{(N)}\})$

1. All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random element $r \in \mathbb{K}$. All parties locally compute
$$x := x^{(1)} + x^{(2)} \cdot r + \ldots + x^{(N)} \cdot r^{N-1}.$$
2. All parties exchange their results x's and check whether they are the same. If a party P_i receives different x's, P_i aborts.

Lemma 8.2. *If there exists two honest parties who receive different set of values $\{x^{(1)}, \ldots, x^{(N)}\}$, then with overwhelming probability, at least one honest party will abort in the protocol* CHECKCONSISTENCY.

Proof. Suppose P_i, P_j are two honest parties and they receive $\{x^{(1)}, \ldots, x^{(N)}\}$ and $\{\tilde{x}^{(1)}, \ldots, \tilde{x}^{(N)}\}$ respectively. Suppose that there exists $i \in [N]$ such that $x^{(i)} \neq \tilde{x}^{(i)}$. Consider the following two polynomials in $\mathbb{K}$:

$$\begin{aligned} f(r) &= x^{(1)} + x^{(2)} \cdot r + \ldots + x^{(N)} \cdot r^{N-1} \\ \tilde{f}(r) &= \tilde{x}^{(1)} + \tilde{x}^{(2)} \cdot r + \ldots + \tilde{x}^{(N)} \cdot r^{N-1} \end{aligned}$$

Since there exists $i \in [N]$ such that $x^{(i)} \neq \tilde{x}^{(i)}$, $f(\cdot)$ and $\tilde{f}(\cdot)$ are two different polynomials. The number of r such that $f(r) = \tilde{f}(r)$ is bounded by the degree of $f(\cdot) - \tilde{f}(\cdot)$, which is $N - 1$. Since r is uniformly chosen from $\mathbb{K}$, the probability that $f(r) = \tilde{f}(r)$ is at most $\frac{N-1}{|\mathbb{K}|} \leq \frac{N-1}{2^\kappa}$. Therefore, with overwhelming probability, $f(r) \neq \tilde{f}(r)$, which means that P_i, P_j will receive different values from each other and abort in the protocol CHECKCONSISTENCY. □

Main Protocol. The protocol MAIN-ROUND-MAL appears in Protocol 8.3.

Lemma 8.3. *Let* c *be the number of clients and* $n = 2t + 1$ *be the number of parties. The protocol* MAIN-ROUND-MAL *securely computes the ideal functionality* $\mathcal{F}_{\textit{main-mal}}$ *in the* $\{\mathcal{F}_{\textit{rand}}, \mathcal{F}_{\textit{zero}}, \mathcal{F}_{\textit{doubleRand}}, \mathcal{F}_{\textit{coin}}, \mathcal{F}_{\textit{multVerify}}\}$*-hybrid model in the presence of a fully malicious adversary controlling up to* c *clients and* t *parties.*

Proof. As we argued in Chapter 2, it is sufficient to focus on adversaries who control exactly t parties. The correctness of MAIN-ROUND-MAL follows from the description.

Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties and honest clients. Recall that $\mathcal{C}orr$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation for MAIN-ROUND-MAL. We describe the strategy of $\mathcal{S}$ phase by phase.

- Simulation for Input Phase:
  For an input x belongs to an honest client, $\mathcal{S}$ randomly samples t elements as the shares of $[x]_t$ of corrupted parties. Then $\mathcal{S}$ sends these shares to corrupted parties.
  For an input x belongs to a corrupted client, $\mathcal{S}$ receives from the adversary the shares held by honest parties. Note that, $\mathcal{S}$ learns $t + 1$ shares of $[x]_t$. $\mathcal{S}$ reconstructs the whole sharing $[x]_t$ and sends x as the input of this corrupted client to $\mathcal{F}_{\text{main-mal}}$.
  Note that for each input sharing, $\mathcal{S}$ learns the shares held by corrupted parties.
- Simulation for Evaluation Phase – Preparing Correlated Randomness:
  In this part, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{zero}}, \mathcal{F}_{\text{doubleRand}}$ and receives the shares of corrupted parties. $\mathcal{S}$ follows the protocol EXPANDZERO and EXPAND to compute the shares of corrupted parties.
- Simulation for Evaluation Phase – Evaluating Two-Layer Circuits:
  In this part, $\mathcal{S}$ simulates the behaviors of honest parties in EVALUATE for each sub-circuit. For each sub-circuit with all the input sharings prepared, $\mathcal{S}$ will know the shares held by corrupted parties. Note that this is true for the first sub-circuit. We describe the strategy of $\mathcal{S}$ step by step.

Figure 8.3: Protocol MAIN-ROUND-MAL

1. **Input Phase**:

   Let $\mathsf{Client}_1, \ldots, \mathsf{Client}_c$ denote the clients who provide inputs. For every input gate of Client_i, Client_i samples a random degree-t Shamir sharing of its input, $[x]_t$, and distributes the shares to all parties.

2. **Evaluation Phase – Preparing Correlated Randomness**:

   All parties start with holding a degree-t sharing for each input gate. The circuit is partitioned into a sequence of two-layer sub-circuits. Let N_1 denote the number of multiplications in the first layer of all sub-circuits, and N_2 denote the number of multiplications in the second layer of all sub-circuits. All parties prepare the correlated randomness as follows:

   - All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare N_1 random degree-t Shamir sharings. Then all parties invoke $\mathcal{F}_{\text{zero}}$ to prepare $N_1 \cdot t/n$ random degree-$2t$ Shamir sharings of 0, and invoke EXPANDZERO to obtain N_1 degree-$2t$ Shamir sharings of 0. These sharings are transformed to N_1 pairs of sharings in the form of $([r]_t, [o]_{2t}, P_i)$.
   - All parties invoke $\mathcal{F}_{\text{doubleRand}}$ to prepare $N_2 \cdot t/n$ pairs of random double sharings. Then all parties invoke EXPAND to obtain N_2 pairs of double sharings in the form of $([r]_t, [r]_{2t}, P_i)$.

3. **Evaluation Phase – Evaluating Two-Layer Circuits**:

   All sub-circuits are evaluated in a predetermined topological order. For each sub-circuit with all the input sharings prepared, all parties invoke EVALUATE to compute the output sharings.

4. **Verification Phase**:

   - Suppose $e^{(1)}, \ldots, e^{(N_1)}$ are the values all parties received in IMPROVED-DN-MULT invoked in EVALUATE. All parties invoke CHECKCONSISTENCY to check that they receive the same values.
   - Suppose $\{([x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^{N_1}$ denote the multiplication tuples computed by IMPROVED-DN-MULT invoked in EVALUATE, and $\{([a^{(i)}]_t, [b^{(i)}]_t, [c^{(i)}]_t)\}_{i=1}^{N_2}$ denote the multiplication tuples computed by MULT invoked in EVALUATE. All parties invoke $\mathcal{F}_{\text{multVerify}}$ to check the correctness of these $N_1 + N_2$ multiplication tuples.

5. **Output Phase**:

   For each output gate of Client_i, let $[x]_t$ denote the input sharing. All parties send their shares of $[x]_t$ to Client_i. Client_i checks whether the shares of $[x]_t$ is consistent. If not, Client_i aborts. Otherwise, Client_i reconstructs the result x.

- In the first step, $\mathcal{S}$ follows the protocol to compute the shares of corrupted parties for the input sharings of each multiplication gate in the second layer.

- In the second step, $\mathcal{S}$ simulates IMPROVED-DN-MULT and MULT as follows. For IMPROVED-DN-MULT:
  - Let $[x]_t, [y]_t$ denote the input sharings. $\mathcal{S}$ has computed the shares of $[x]_t, [y]_t$ held by corrupted parties. Let $([r]_t, [o]_{2t}, P_i)$ denote the sharings associated with this gate. $\mathcal{S}$ learns the shares of $[r]_t, [o]_{2t}$ held by corrupted parties when emulating $\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{zero}}$ and simulating EXPANDZERO.
  - $\mathcal{S}$ computes the shares of $[e]_{2t} := [x]_t \cdot [y]_t + [r]_t + [o]_{2t}$ held by corrupted parties. If P_i is corrupted, $\mathcal{S}$ samples random elements as shares of $[e]_{2t}$ of honest parties.
  - Depending on whether P_i is honest, there are two cases:
    - If P_i is honest, $\mathcal{S}$ receives the shares from corrupted parties. Let $[\tilde{e}]_{2t}$ denote the sharing when using the shares received from corrupted parties. This is to distinguish from $[e]_{2t}$ which uses the shares computed by $\mathcal{S}$ for corrupted parties. Then $\mathcal{S}$ computes the shares of $[d]_{2t} := [\tilde{e}]_{2t} - [e]_{2t}$ held by corrupted parties. Note that the shares of $[d]_{2t}$ held by honest parties are 0. $\mathcal{S}$ reconstructs the secret d. Finally, $\mathcal{S}$ samples a random element as e and sends e to corrupted parties.
    - If P_i is corrupted, $\mathcal{S}$ sends the shares of $[e]_{2t}$ of honest parties to P_i. Note that for $[e]_{2t}$, $\mathcal{S}$ knows all the shares. $\mathcal{S}$ reconstructs the secret e. Then $\mathcal{S}$ receives the value $\tilde{e}$ from P_i. If P_i sends different values to different honest parties, $\mathcal{S}$ marks this execution as `fail`, and uses the value of the first honest party. Finally, $\mathcal{S}$ computes $d := \tilde{e} - e$.
  - In the last step, $\mathcal{S}$ computes the shares of $[z]_t := \tilde{e} - [r]_t$ held by corrupted parties.

  For MULT:
  - Let $[x]_t, [y]_t$ denote the input sharings. $\mathcal{S}$ has computed the shares of $[x]_t, [y]_t$ held by corrupted parties. Let $([r]_t, [r]_{2t}, P_i)$ denote the sharings associated with this gate. $\mathcal{S}$ learns the shares of $[r]_t, [r]_{2t}$ held by corrupted parties when emulating $\mathcal{F}_{\text{doubleRand}}$ and simulating EXPAND.
  - $\mathcal{S}$ computes the shares of $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$ held by corrupted parties. If P_i is corrupted, $\mathcal{S}$ samples random elements as shares of $[e]_{2t}$ of honest parties.
  - Depending on whether P_i is honest, there are two cases:
    - If P_i is honest, $\mathcal{S}$ receives the shares from corrupted parties. Let $[\tilde{e}]_{2t}$ denote the sharing when using the shares received from corrupted parties. This is to distinguish from $[e]_{2t}$ which uses the shares computed by $\mathcal{S}$ for corrupted parties. Then $\mathcal{S}$ computes the shares of $[d]_{2t} := [\tilde{e}]_{2t} - [e]_{2t}$ held by corrupted parties. Note that the shares of $[d]_{2t}$ held by honest parties are 0. $\mathcal{S}$ reconstructs the secret d. Finally, for $[e]_t$, $\mathcal{S}$ samples random elements as the shares of corrupted parties.
    - If P_i is corrupted, $\mathcal{S}$ sends the shares of $[e]_{2t}$ of honest parties to P_i. Note that for $[e]_{2t}$, $\mathcal{S}$ knows all the shares. $\mathcal{S}$ reconstructs the secret e. Then $\mathcal{S}$ receives the shares of $[e]_t$ of honest parties from P_i. $\mathcal{S}$ reconstructs the whole sharing and computes the secret of $[e]_t$, denoted by $\tilde{e}$. Finally, $\mathcal{S}$ computes $d := \tilde{e} - e$.

- In the last step, $\mathcal{S}$ computes the shares of $[z]_t := [e]_t - [r]_t$ held by corrupted parties.
  - In the rest of two steps (which only contain local computation), $\mathcal{S}$ follows the protocol and computes the shares of corrupted parties for each degree-t Shamir sharing.
- Simulation for Verification Phase:
  - $\mathcal{S}$ first simulates CHECKCONSISTENCY. Note that $e^{(1)}, \ldots, e^{(N_1)}$ are either received from corrupted $P_{\texttt{king}}$'s or explicitly generated by $\mathcal{S}$. $\mathcal{S}$ follows the protocol honestly. If any party aborts *or $\mathcal{S}$ has marked this execution as* `fail`, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{main-mal}}$ and aborts.
  - $\mathcal{S}$ then emulates the functionality $\mathcal{F}_{\text{multVerify}}$. Recall that in the simulation of the computation phase, $\mathcal{S}$ has computed the shares of each multiplication tuple held by corrupted parties and the difference. $\mathcal{S}$ directly sends these shares and differences to the adversary as in $\mathcal{F}_{\text{multVerify}}$. If there exists a non-zero difference, $\mathcal{S}$ sets $b = \texttt{abort}$. Otherwise, $\mathcal{S}$ sets $b = \texttt{accept}$. Then $\mathcal{S}$ sends b to the adversary.
    - If $b = \texttt{accept}$ and the adversary replies `continue`, $\mathcal{S}$ moves to the next phase.
    - Otherwise, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{main-mal}}$ and aborts.
- Simulation for Output Phase:
  For each output gate with $[x]_t$ associated with it, if the receiver is an honest client, $\mathcal{S}$ receives from the adversary the shares held by corrupted parties. Then $\mathcal{S}$ checks whether the shares are the same as the ones computed by $\mathcal{S}$. If true, $\mathcal{S}$ accepts the output. Otherwise, $\mathcal{S}$ rejects the output.
  If the receiver is a corrupted client, $\mathcal{S}$ receives the result x from $\mathcal{F}_{\text{main-mal}}$. Then, based on the shares of $[x]_t$ held by corrupted parties and the secret x, $\mathcal{S}$ reconstructs the shares held by honest parties, and sends these shares to the adversary.
  Finally, if $\mathcal{S}$ rejects any output of honest clients, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{main-mal}}$. Otherwise, $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\text{main-mal}}$.

Hybrid Arguments. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties with overwhelming probability. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ computes the input of corrupted clients and sends them to $\mathcal{F}_{\text{main-mal}}$. The distribution of **Hybrid**$_1$ is identical to **Hybrid**$_0$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ simulates CHECKCONSISTENCY. Concretely, $\mathcal{S}$ checks whether all honest parties receive the same values from $P_{\texttt{king}}$ in IMPROVED-DN-MULT. If not, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{main-mal}}$ and aborts. According to Lemma 8.2, the probability that an honest party aborts in this case is overwhelming. Therefore, the distribution of **Hybrid**$_2$ is statistically close to the distribution of **Hybrid**$_1$.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ simulates the output phase as described above. Note that the output phase is executed after the verification phase, which ensures the correctness of IMPROVED-DN-MULT and MULT. Therefore, the output phase is executed only when the computation is correct. For an output gate with $[x]_t$ associated with it, if the receiver is an honest client, the shares that corrupted parties should hold are determined by the shares of honest parties. Therefore, if corrupted parties send different shares from the ones computed by $\mathcal{S}$, the shares of $[x]_t$

will be inconsistent and the client will reject the output. If the receiver is a corrupted client, the shares of honest parties are determined by the output x and the shares held by corrupted parties. Therefore, the shares prepared by $\mathcal{S}$ are identical to the real shares held by honest parties.

Therefore, the distribution of $\mathbf{Hybrid}_3$ is identical to the distribution of $\mathbf{Hybrid}_2$.

$\mathbf{Hybrid}_4$: In this hybrid, $\mathcal{S}$ computes the difference of each multiplication tuple in the computation phase. Then $\mathcal{S}$ simulates the verification phase. Note that $\mathcal{F}_{\text{multVerify}}$ simply checks whether there is an incorrect multiplication tuple, which is equivalent to check whether there is a non-zero difference.

Therefore, the distribution of $\mathbf{Hybrid}_4$ is identical to the distribution of $\mathbf{Hybrid}_3$.

$\mathbf{Hybrid}_5$: In this hybrid, $\mathcal{S}$ simulates EVALUATE. Since the parts which require interaction are IMPROVED-DN-MULT and MULT, we only focus on the simulation of these two protocols. We argue the following two points:

- When a corrupted party P_i behaves as $P_{\texttt{king}}$, the shares of $[e]_{2t}$ of honest parties are uniformly random in both IMPROVED-DN-MULT and MULT.
- When $P_{\texttt{king}}$'s send the same values to all honest parties in IMPROVED-DN-MULT, $\mathcal{S}$ extracts the correct difference for each multiplication tuple computed by IMPROVED-DN-MULT and MULT.

For the first point, following from a similar argument in Lemma 6.1, the random sharing $[r]_t + [o]_{2t}$ in IMPROVED-DN-MULT and the random sharing $[r]_{2t}$ in MULT satisfy that the shares of honest parties are uniformly random. Therefore, the shares of $[e]_{2t}$ of honest parties are uniformly random. For the second point, when $P_{\texttt{king}}$'s send the same values to all honest parties, all degree-t Shamir sharings are consistent in the sense that the shares of corrupted parties computed by $\mathcal{S}$ are consistent with the shares of honest parties. In this case, following from a similar argument in Lemma 6.1, $\mathcal{S}$ extracts the correct difference for each multiplication tuple computed by IMPROVED-DN-MULT and MULT.

Note that $\mathcal{F}_{\text{multVerify}}$ is only executed when all parties receive the same values from $P_{\texttt{king}}$'s in IMPROVED-DN-MULT. Therefore, the above two points show that the distribution of $\mathbf{Hybrid}_5$ is identical to the distribution of $\mathbf{Hybrid}_4$.

$\mathbf{Hybrid}_6$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{doubleRand}}, \mathcal{F}_{\text{zero}}$ and simulates EXPANDZERO and EXPAND as described above. Note that only the shares of corrupted parties are used in the rest of steps in $\mathbf{Hybrid}_5$. Therefore, the distribution of $\mathbf{Hybrid}_6$ is identical to the distribution of $\mathbf{Hybrid}_5$.

$\mathbf{Hybrid}_7$: In this hybrid, $\mathcal{S}$ simulates the input phase. The only difference between $\mathbf{Hybrid}_6$ and $\mathbf{Hybrid}_7$ is that, in $\mathbf{Hybrid}_6$, $\mathcal{S}$ uses the real input of honest clients to generate the input sharings, while in $\mathbf{Hybrid}_7$, $\mathcal{S}$ simply samples random elements as the shares of corrupted parties. Note that the distributions of the shares of corrupted parties in both hybrids are the same. Therefore, the distribution of $\mathbf{Hybrid}_7$ is the same as $\mathbf{Hybrid}_6$.

Note that $\mathbf{Hybrid}_7$ is the execution in the ideal world, and the distribution of $\mathbf{Hybrid}_7$ is identical to the distribution of $\mathbf{Hybrid}_0$, the execution in the real world. □

Analysis of the Concrete Efficiency. Compared with the semi-honest protocol MAIN-ROUND, the maliciously secure protocol MAIN-ROUND-MAL only invokes CHECKCONSISTENCY and

$\mathcal{F}_{\text{multVerify}}$ to check the correctness of the computation. Recall that CHECKCONSISTENCY requires $O(n^2 \cdot \kappa)$ bits. And when using our fast verification protocol MULTVERIFY to instantiate $\mathcal{F}_{\text{multVerify}}$, it requires $O(n^2 \cdot \kappa + n \cdot \kappa^2)$ bits (when setting the compression factor $k = \kappa$ in MULTVERIFY). Regarding the round complexity of the verification phases, note that both CHECKCONSISTENCY and MULTVERIFY only need constant number of rounds. Thus, we have the following theorem.

Theorem 1.2. *Let n be a positive integer and $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq n + 1$. For all arithmetic circuit C over $\mathbb{F}$, there is an information-theoretic n-party MPC protocol, which achieves malicious security (with abort) against a fully malicious adversary controlling $t < n/2$ corrupted parties, with communication complexity $O(|C| \cdot n + n^2 \cdot \kappa + n \cdot \kappa^2)$ elements, where κ is the security parameter and $|C|$ is the circuit size. Furthermore, the concrete efficiency is 4.5 elements per party per multiplication gate but halving the number of rounds (up to a constant number of rounds).*

Part II

Sharing Transformation and Applications to Dishonest Majority MPC with Packed Secret Sharing

Chapter 9

Introduction for Part II

In this part we initiate the study of *sharing transformations* which allow us to perform *arbitrary* linear maps on the secrets of (possibly packed) secret-sharing schemes. More specifically, suppose Σ and Σ' are two linear secret sharing schemes over a finite field $\mathbb{F}$. A set of n parties $\{P_1, P_2, \ldots, P_n\}$ start with holding a Σ-sharing $\boldsymbol{X}$. Here $\boldsymbol{X}$ could be the sharing of a single field element or a vector of field elements (e.g., as in packed secret sharing where multiple secrets are stored within a single sharing). The parties wish to compute a Σ'-sharing $\boldsymbol{Y}$ whose secret is a *linear map* of the secret of $\boldsymbol{X}$. Here a linear map means that each output secret is a linear combination of the input secrets (recall that the secret can be a vector in $\mathbb{F}$). We refer to this problem as *sharing transformation*.

Restricted cases of sharing transformations occur frequently in the construction of secure computation protocols based on secret sharing. For example,

- In the well-known BGW protocol [9] and DN protocol [28] and their followups (see [17, 24, 45] and the citations therein), when evaluating a multiplication gate, all parties first locally compute a Shamir secret sharing of the result with a larger degree. To proceed the computation, all parties wish to transform it to a Shamir secret sharing of the result with a smaller degree. Here the two linear secret sharing schemes Σ, Σ' are both the Shamir secret sharing schemes but with different degrees.
- A recent line of works [21, 27, 62] use the notion of reverse multiplication-friendly embeddings (RMFE) to construct efficient information-theoretic MPC protocols over small fields or rings $\mathbb{Z}/p^\ell\mathbb{Z}$. This technique requires all parties to transform a secret sharing of a vector of secrets that are encoded by an encoding scheme to another secret sharing of the same secrets that are encoded by a different encoding scheme.
- A line of works [7, 29, 39, 41] focus on the strong honest majority setting (i.e., $t = (1/2 - \epsilon) \cdot n$) and use the packed secret-sharing technique [34] to construct MPC protocols with sub-linear communication complexity in the number of parties. The main technical difficulty is to perform a linear map on the secrets of a single packed secret sharing (e.g., permutation or fan-out). In particular, depending on the circuit, each time the linear map we need to perform can be different.

Unlike the above results, our sharing transformation protocol (1) can perform arbitrary linear maps (2) is not restricted to a specific secret-sharing scheme and (3) can achieve optimal commu-

nication complexity[1]. Our transformation can find applications to different protocols based on different secret sharing schemes. In this part we focus on applications to information-theoretic (IT) MPC protocols. Furthermore, since we can handle any linear secret sharing scheme, our sharing transformation works for an arbitrary packing factor k as long as $t \leq n - 2k + 1$ where n is the number of participants and t is the number of corrupted parties by the adversary. This allows us to present the first IT MPC protocols with online communication complexity per gate sub-linear in the number of parties in the circuit-independent prepossessing model for a variety of corruption thresholds based on packed secret sharing.

Previous Results in the Dishonest Majority Setting. In the dishonest majority setting, preprocessing model is required for the existence of information-theoretic MPC protocols. For the case where $t = n - 1$, any function can be computed with IT security in the preprocessing model with online communication complexity of $O(n)$ field elements per gate across all parties [30]. Existing protocols in the literature even for $t \in [(n + 1)/2, n - 1]$ still required communication complexity of $O(n)$ elements per gate.

Previous Results in the (Strong) Honest Majority Setting. In the (strong) honest majority setting, information-theoretic MPC protocols exist in the plain model. The best known protocols in the so called optimal threshold regime tolerating $t = (n - 1)/2$ corrupted parties require communicating $O(n)$ field elements per gate (ignoring circuit independent terms) [13, 17, 24, 28, 38, 44, 60]. There are no constructions known beating this barrier even in the semi-honest setting despite over a decade of research.

In a remarkable result, Damgård et al. [29] showed an unconditional MPC protocol with communication complexity of $O(\log |C| \cdot 1/\epsilon)$ per gate (ignoring circuit independent terms) tolerating $t = (1/3 - \epsilon) \cdot n$ corrupted parties. This was later extended by Genkin et al. [39] to obtain a construction tolerating $t = (1/2 - \epsilon) \cdot n$ corrupted parties with also a constant factor improvement in the communication complexity. These works rely on the packed secret sharing technique introduced by Franklin and Yung [34] where k secrets are packed into a single secret sharing. An incomparable result was given by Garay et al. [37] who obtained a protocol with communication complexity $O(\log^{1+\delta} n)$ per gate where δ is any positive constant.

Two recent independent works [7, 41] also use the packed secret sharing technique in the MPC paradigm. The work [41] uses the packed secret sharing and achieves $O(1)$ elements per gate in the preprocessing phase. However, it still requires $O(n)$ elements per gate in the online phase. On the other hand, the work [7] achieves $O(1)$ elements per gate only for a special class of circuits that have a highly repetitive structure.

[1]To be more precise, our protocol achieves linear communication complexity in the summation of the sharing sizes of the two secret sharing schemes in the transformation. This is optimal (up to a constant factor) since it matches the communication complexity of using an ideal functionality to do sharing transformation: the size of the input is the sharing size of the first secret sharing scheme, the size of the output is the sharing size of the second secret sharing scheme, and the communication complexity is the size of the input and output.

9.1 Our Contributions

Sharing Transformation. For our arbitrary linear-map transformation on (packed) linear secret sharing schemes we obtain the following informal result focusing on share size 1 (i.e., each share is a single field element).

Theorem 9.1 (Informal)**.** *Let $k = (n - t + 1)/2$. For all k tuples of $\{(\Sigma_i, \Sigma'_i, f_i)\}_{i=1}^{k}$ linear secret sharing schemes with injective sharing functions and for all Σ_i-sharings $\{\boldsymbol{X}_i\}_{i=1}^{k}$, there is an information-theoretic MPC protocol with semi-honest security against t corrupted parties that transforms $\boldsymbol{X}_i$ to a Σ'_i-sharing $\boldsymbol{Y}_i$ such that the secret of $\boldsymbol{Y}_i$ is equal to the result of applying a linear map f_i on the secret of $\boldsymbol{X}_i$ for all $i \in \{1, \ldots, k\}$ (Here the secrets of $\boldsymbol{X}_i$ and $\boldsymbol{Y}_i$ can be vectors). The cost of the protocol is $O(n^3/k^2)$ elements of communication per sharing in a (sharing independent) preprocessing stage leading to preprocessed data of size $O(n^2/k)$, and $O(n^2/k)$ elements of communication per sharing in the online phase. When $t = (1 - \epsilon) \cdot n$ for a positive constant ϵ, the overall communication complexity is $O(n)$ elements per sharing transformation.*

The formal theorem is stated in Theorem 11.1. In Chapter 11, we show that our sharing transformation works for any share size ℓ (with an increase in the communication complexity by a factor ℓ), and is naturally extended to any finite fields and rings $\mathbb{Z}/p^{\ell}\mathbb{Z}$. The main application of our sharing transformation technique is to construct MPC protocols. And we achieve malicious security by directly compiling our semi-honest MPC protocol instead of relying on a maliciously secure sharing transformation protocol. Therefore, we do not attempt to achieve malicious security for our sharing transformation technique.

We now turn our attention to constructing general MPC using our sharing transformation technique.

Dishonest Majority Setting. In the setting of dishonest majority where the number of corrupted parties $t = (1 - \epsilon) \cdot n$ for a positive constant ϵ, our MPC protocol achieves the cost of $O(1/\epsilon^2)$ elements of (the size of) preprocessing data, and $O(1/\epsilon)$ elements of communication per gate among all parties. Thus when ϵ is a constant (e.g., up to 99 percent of all parties may be corrupted), the achieved communication complexity in the online phase is $O(1)$ elements per gate.

Honest Majority Setting. As a corollary of our results in the dishonest majority setting, we can achieve $O(1)$ elements per gate of online communication and $O(1)$ elements of preprocessing data per gate across all parties in the honest majority setting. In particular, the preprocessing data can be efficiently prepared relying on our efficient multiplication protocol and efficient multiplication verification protocol in Part I. As a result, we obtain an information-theoretic MPC protocol with $O(n)$ elements of communication per gate in the preprocessing phase, and $O(1)$ elements of communication per gate in the online phase.

Our main results are summarized below. Note that we have omitted the additive terms of the overhead of the communication complexity in the informal theorems below. The formal theorems are stated in Theorem 12.1 and Theorem 14.1 respectively, where the additive terms

are dependent on n and the depth of the evaluated circuit. Our first theorem is for the semi-honest setting:

Theorem 1.3 (Informal)**.** *For an arithmetic circuit C over a finite field $\mathbb{F}$ of size $|\mathbb{F}| \geq |C| + n$, there exists an information-theoretic MPC protocol in the preprocessing model which securely computes the arithmetic circuit C in the presence of a semi-honest adversary controlling up to t parties. The cost of the protocol is $O(|C| \cdot n^2/k^2)$ elements of preprocessing data, and $O(|C| \cdot n/k)$ elements of communication where $k = \frac{n-t+1}{2}$ is the packing parameter. For the case where $k = O(n)$, the achieved communication complexity in the online phase is $O(1)$ elements per gate.*

Our theorem also holds in the presence of a malicious adversary for all $\frac{n-1}{3} \leq t \leq n-1$.

Theorem 1.4 (Informal)**.** *For an arithmetic circuit C over a finite field $\mathbb{F}$ of size $|\mathbb{F}| \geq 2^\kappa$, where κ is the security parameter, and for all $\frac{n-1}{3} \leq t \leq n-1$, there exists an information-theoretic MPC protocol in the preprocessing model which securely computes the arithmetic circuit C in the presence of a fully malicious adversary controlling up to t parties. The cost of the protocol is $O(|C| \cdot n^2/k^2)$ elements of preprocessing data, and $O(|C| \cdot n/k)$ elements of communication where $k = \frac{n-t+1}{2}$ is the packing parameter. For the case where $k = O(n)$, the achieved communication complexity in the online phase is $O(1)$ elements per gate.*

Moreover, we also propose an alternative solution to achieve the same results for small finite fields of size $|\mathbb{F}| \geq 2n$ based on the Hall's Marriage Theorem in the graph theory, which may be of independent interest. See more details in Chapter 10.2.4 and Chapter 13.

9.2 Related Works

Sharing Transformation and Sharing Conversion. In this work, we study the problem of sharing transformation, where we want to transform the shares of one or multiple secrets under one secret sharing scheme into shares of another secret sharing scheme and apply a function on the secrets. In particular, we require the two secret sharing schemes as well as the function to be linear in the same finite commutative ring.

On the other hand, a line of works [3, 15, 32, 33, 47, 58, 61, 63] study the problem of sharing conversion where they focus on the identity function (i.e., keep the secret unchanged) but two secret sharing schemes that are not in the same finite commutative ring, e.g., converting a secret sharing in the binary field to a secret sharing in a prime field. The problem of sharing conversion appears to be much difficult than the problem of sharing transformation since we need to handle two different rings or fields. In particular, our technique does not work for the problem of sharing conversion.

Sharing Transformation via Pseudo-Random Secret Sharing [12]. The work [12] also studies the problem of sharing transformation relying on the technique of PRSS (Pseudo-Random Secret Sharing) initially studied in [26]. At a high level, the idea is to first prepare replicated secret sharings and then transform them to correlated random packed Shamir sharings, which are used for sharing transformations.

The main advantage of this approach is that, after some initial setup, replicated secret sharings can be prepared without interaction relying on pseudo-random generators. Relying on this

technique, the work [12] can realize network routing, a key step of using the packed secret sharing technique in MPC, with no extra cost.

However, both the communication complexity for the setup and the computation complexity for generating each replicated secret sharing grows exponentially with the number of corrupted parties. It restricts the technique in [12] to be only practical for a small constant number of parties.

Information-Theoretic MPC with Dishonest Majority. In [30], Damgård et al. introduced the well-known protocol SPDZ in the all-but-one corruption setting, i.e., the number of corrupted parties is $t = n - 1$. The online phase of SPDZ is information-theoretic and therefore can be viewed as an information-theoretic protocol in the preprocessing model. The cost of the SPDZ protocol is $O(n)$ elements of preprocessing data and $O(n)$ elements of communication per multiplication gate among all parties. Also in the all-but-one corruption setting, a recent breakthrough [25] shows that information-theoretic protocols in the preprocessing model are possible to achieve with sublinear communication complexity in the *circuit size* at a cost of a large amount of preprocessing data. Concretely, Couteau presents an MPC protocol for *layered circuits* in the preprocessing model, which achieves communication complexity of $O(n \cdot |C|/\log\log\log|C|)$ elements, at the cost of $O(|C|^2/\log\log\log|C|)$ elements of preprocessing data. We note that, however, the communication complexity of the protocol in [25] is still linear in the number of parties and the protocol is impractical for a large circuit due to the amount of preprocessing data. In fact, Ishai et al. [51] showed that it is possible to achieve a communication complexity that is only linear in the input size and independent of the circuit size at the cost of an exponential amount of preprocessing data in the input size.

Compared with [25, 30], we focus on a general corruption threshold where the number of corrupted parties $t = (1 - \epsilon) \cdot n$ for a positive constant ϵ. Our protocol achieves the cost of $O(1)$ elements of preprocessing data and $O(1)$ elements of communication per gate among all parties. One advantage of our protocol is that one may trade the corruption threshold with the real speed-up in the protocol, which is otherwise not possible in [25, 30].

We note that a folklore solution in our setting is to choose a random small committee and then evaluate the circuit among parties in the small committee. Since there are $\epsilon \cdot n$ honest parties, if each time we add a random party into the committee, the probability of selecting an honest party is ϵ. Let κ be the security parameter. To ensure that with probability $1 - 2^{-\kappa}$, there is at least one honest party in the committee, the size of the committee should be at least $O(\kappa)$. If parties in the selected committee run the protocol in [30], the achieved cost is $O(\kappa)$ elements of preprocessing data and $O(\kappa)$ elements of communication per gate among all parties, which is κ times of the cost of our construction. If parties run the protocol in [25], the achieved cost is $O(|C|/\log\log\log|C|)$ elements of preprocessing data and $O(\kappa/(\epsilon \cdot \log\log\log|C|))$ elements of communication per gate among all parties. Note that, however, the circuit size is bounded by a polynomial of the security parameter κ, which means the term $\log\log\log|C|$ is much smaller than κ. Therefore, no matter which protocols are used in the folklore solution, our protocol is always $o(\kappa)$ times faster.

Instantiation of Beaver Triples in the Preprocessing Phase. A line of works focus on generating the preprocessed data (i.e., Beaver triples) for SPDZ-style protocols based on various cryptographic assumptions. For example, the works [30, 54] use somewhat homomorphic encryption schemes and achieve $O(n^2)$ communication per Beaver triple. The work [53] uses OT and also achieves $O(n^2)$ communication per Beaver triple. A recent breakthrough by Boyle, et al [14] (with a concrete efficient instantiation in [16]) shows that it is possible to achieve a sublinear communication complexity in the number of Beaver triples based on the assumption of LWE or Ring-LPN.

On the other hand, our construction requires to prepare *packed* Beaver triples in the preprocessing phase. We leave the question of extending the techniques in [14, 16, 30, 53, 54] to prepare packed Beaver triples to future works.

Information-Theoretic MPC with Strong Honest Majority. In the setting of strong honest majority setting where the number of corrupted parties $t = (1/2 - \epsilon) \cdot n$ for a positive constant ϵ, a rich line of works [7, 12, 29, 39, 41, 46, 52] makes use of the packed secret sharing technique to construct efficient multiparty computation protocols. In particular, the recent work [46] gives the first information-theoretic MPC protocol in this setting which achieves $O(1)$ communication complexity per gate among all parties.

Communication Complexity versus Computation Complexity. In the dishonest majority setting with corruption threshold $t = (1 - \epsilon) \cdot n$ for a positive constant ϵ, our work achieves $O(|C|)$ total amount of preprocessing data and online communication. However, our construction has online *computation* complexity $O(|C| \cdot n)$, which grows linearly with the number of parties. The linear computation complexity is due to the network routing: when all parties collect secrets from multiple packed sharings, the computation complexity grows linearly with the number of different packed sharings. It means that our use of packed secret sharing does not help reduce the computation complexity. Also, our sharing transformation protocol requires linear computation complexity due to a similar reason.

In the strong honest majority setting, the work [29] achieves sublinear communication (but with a $\log |C|$ factor compared with our work) and *computation* complexity in the number of parties. We may achieve a similar result in our setting by the following potential approaches:

- We may directly extend the network routing protocol in [29] to our setting.

  At a high level, the work [29] transforms a general circuit to a SIMD circuit by using the Beneš network so that only a limited number of different sharing transformations is required to perform. The transformation increases the circuit size by a $\log |C|$ factor. The use of Beneš network has two advantages:

  1. First, after the transformation, the computation only requires to perform a limited number of different sharing transformations. Therefore, the generic approach of doing sharing transformation (which is only efficient when the same transformation is performed multiple times) suffices.
  2. Second, there is no need to do network routing for a SIMD circuit (except the inserted sharing transformations due to the use of Beneš network).

These two advantages allow them to also achieve sublinear computation complexity.

In our setting, we may apply the same circuit transformation, use the generic approach for sharing transformations, and then use our protocol to evaluate the circuit.

- We may combine the technique of party virtualization [19] and the protocol in [29] as follows:
  - We first randomly select N random constant size committees. Each committee simulates one virtual party by using a maximal threshold dishonest majority protocol that has linear overhead in the circuit size (such as the SPDZ protocol). Note that if a committee contains at least one honest party, the corresponding virtual party behaves honestly. By carefully adjusting the size of the committee and the number of committees, we may ensure that with overwhelming probability, over $2/3$ of virtual parties are honest.
  - Then the N virtual parties run the protocol in [29], which achieves sublinear communication and computation complexity in the number of virtual parties.

  Now we analyze the computation complexity of the above approach. Note that the circuit for simulating a virtual party is proportional to the computation complexity of this party. Since each virtual party is simulated by a constant size committee, the simulation of a virtual party only increases the computation complexity by a constant factor. Therefore the achieved computation complexity remains sublinear in the number of parties. (Note that the communication complexity is always smaller than the computation complexity.)

We leave the formal descriptions and verifications of the above two approaches to future works.

Chapter 10

Technical Overview

In this chapter, we give an overview of our techniques. We use bold letters to represent vectors.

Reducing Sharing Transformation to Random Sharing Preparation. Usually, sharing transformation is solved by using a pair of random sharings $(\boldsymbol{R}, \boldsymbol{R}')$ such that $\boldsymbol{R}$ is a random Σ-sharing and $\boldsymbol{R}'$ is a random Σ'-sharing which satisfies that the secret of $\boldsymbol{R}'$ is equal to the result of applying f on the secret of $\boldsymbol{R}$, where f is the desired linear map. Then all parties can run the following steps to efficiently transform $\boldsymbol{X}$ to $\boldsymbol{Y}$.

1. All parties locally compute $\boldsymbol{X} + \boldsymbol{R}$ and send their shares to the first party P_1.
2. P_1 reconstructs the secret of $\boldsymbol{X}+\boldsymbol{R}$, denoted by $\boldsymbol{w}$. Then P_1 computes $f(\boldsymbol{w})$ and generates a Σ'-sharing of $f(\boldsymbol{w})$, denoted by $\boldsymbol{W}$. Finally, P_1 distributes the shares of $\boldsymbol{W}$ to all parties.
3. All parties locally compute $\boldsymbol{Y} = \boldsymbol{W} - \boldsymbol{R}'$.

If we use $\mathtt{rec}, \mathtt{rec}'$ to denote the reconstruction maps of Σ and Σ' (which are linear by definition) respectively, the correctness follows from that

$$\mathtt{rec}'(\boldsymbol{Y}) = \mathtt{rec}'(\boldsymbol{W}) - \mathtt{rec}'(\boldsymbol{R}') = f(\boldsymbol{w}) - f(\mathtt{rec}(\boldsymbol{R})) = f(\mathtt{rec}(\boldsymbol{X}+\boldsymbol{R}) - \mathtt{rec}(\boldsymbol{R})) = f(\mathtt{rec}(\boldsymbol{X})).$$

And the security follows from the fact that $\boldsymbol{X} + \boldsymbol{R}$ is a random Σ-sharing and thus reveals no information about the secret of $\boldsymbol{X}$. Therefore, the problem of sharing transformation is reduced to preparing a pair of random sharings $(\boldsymbol{R}, \boldsymbol{R}')$. Let $\widetilde{\Sigma} = \widetilde{\Sigma}(\Sigma, \Sigma', f)$ be the secret sharing scheme which satisfies that a $\widetilde{\Sigma}$-sharing of a secret $\boldsymbol{x}$ consists of $\boldsymbol{X}$ which is a Σ-sharing of $\boldsymbol{x}$, and $\boldsymbol{Y}$ which is a Σ'-sharing of $f(\boldsymbol{x})$. Then, the goal becomes to prepare a random $\widetilde{\Sigma}$-sharing.

The generic approach of preparing random sharings of a linear secret sharing scheme over $\mathbb{F}$ is as follows:

1. Each party P_i first samples a random sharing $\boldsymbol{R}_i$ and distributes the shares to all other parties.
2. All parties use a linear randomness extractor over $\mathbb{F}$ to extract a batch of random sharings such that they remain uniformly random even given the random sharings sampled by corrupted parties. For a large finite field, we can use the transpose of a Vandermonde matrix [28] as a linear randomness extractor. The use of a randomness extractor is to reduce the communication complexity per random sharing. Alternatively, we can simply add all

random sharings $\{\boldsymbol{R}_i\}_{i=1}^n$ and output a single random sharing, which results in quadratic communication complexity in the number of parties.

If t is the number of corrupted parties, all parties can extract $n-t$ random sharings when using a large finite field. Then, the amortized communication cost per sharing is $n^2/(n-t)$ field elements (assuming each share is a single field element). When $n-t=O(n)$, e.g., the honest majority setting, the amortized cost becomes $n^2/(n-t)=O(n)$, which is generally good enough since it matches the communication complexity of delivering a random sharing by a trusted party, which seems like the best we can hope, up to a constant factor.

Thus when we need to prepare many random sharings for the same linear secret sharing scheme, the generic approach is already good enough. And in particular, it is good enough for random $\widetilde{\Sigma}$-sharings which are used for the *same* sharing transformation defined by $\widetilde{\Sigma} = \widetilde{\Sigma}(\Sigma, \Sigma', f)$, since $\widetilde{\Sigma}$ is also a linear secret sharing scheme. This is exactly the case when we need to do degree reduction in [9, 28] and change the encoding of the secrets in [21, 27, 62]. However, it is a different story if we need to prepare random sharings for different linear secret sharing schemes: If only a constant number of random sharings are needed for each linear secret sharing scheme, the amortized cost per sharing becomes $O(n^2)$ field elements. This is exactly the case when we need to perform permutation on the secrets of a packed secret sharing in [7, 29, 39]. In their setting, the permutations are determined by the circuit structure. In particular, these permutations can all be distinct in the worst case. As a result, the cost of preparing random sharings becomes the dominating term in the communication complexity in the MPC protocols. To avoid it, previous works either restrict the number of different secret sharing schemes they need to prepare random sharings for [29, 39] or restrict the types of circuits [7].

This leads to the following fundamental question: *Can we prepare random sharings (used for sharing transformations) for different linear secret sharing schemes with amortized communication complexity $O(n)$?*

10.1 Preparing Random Sharings for Different Linear Secret Sharing Schemes

To better expose our idea, we focus on a large finite field $\mathbb{F}$. In the following, we use n for the number of parties, and t for the number of corrupted parties. We assume semi-honest security in the technical overview.

Linear Secret Sharing Scheme over $\mathbb{F}$. For a linear secret sharing scheme Σ over $\mathbb{F}$, we use $Z=\mathbb{F}^{\tilde{k}}$ to denote the secret space. $\tilde{k}$ is also referred to as the secret size of Σ. For simplicity, we focus on the linear secret sharing schemes that have share size 1 (i.e., each share is a single field element even though the secret is a vector of $\tilde{k}$ elements). Let $\mathtt{share} : Z \times \mathbb{F}^{\tilde{r}} \to \mathbb{F}^n$ be the deterministic sharing map which takes as input a secret $\boldsymbol{x}$ and $\tilde{r}$ random field elements, and outputs a Σ-sharing of $\boldsymbol{x}$. We focus on linear secret sharing schemes whose sharing maps are injective, which implies that $\tilde{k}+\tilde{r} \le n$. Let $\mathtt{rec} : \mathbb{F}^n \to Z$ be the reconstruction map which takes as input a Σ-sharing and outputs the secret of the input sharing. As discussed above, we have shown that preparing many random sharings for the same linear secret sharing scheme can

be efficiently achieved.

We use the standard Shamir secret sharing scheme over $\mathbb{F}$, and use $[x]_t$ to denote a degree-t Shamir sharing of x. A degree-t Shamir sharing requires $t+1$ shares to reconstruct the secret. And any t shares of a degree-t Shamir sharing are independent of the secret.

10.1.1 Starting Point - Preparing a Random Sharing for a Single Linear Secret Sharing Scheme

Let Σ be an arbitrary linear secret sharing scheme. Although we have already shown how to prepare a random sharing for a single linear secret sharing scheme Σ, we consider the following process which is easy to be extended (discussed later).

1. All parties prepare $\tilde{k}+\tilde{r}$ random degree-t Shamir sharings. Let $\boldsymbol{\tau}$ be the secrets of the first $\tilde{k}$ sharings, and $\boldsymbol{\rho}$ be the secrets of the last $\tilde{r}$ sharings. Our goal is to compute a random Σ-sharing of $\boldsymbol{\tau}$ with random tape $\boldsymbol{\rho}$, i.e., $\mathtt{share}(\boldsymbol{\tau}, \boldsymbol{\rho})$.
2. Since $\mathtt{share}$ is $\mathbb{F}$-linear, for all $j \in \{1, 2, \ldots, n\}$, the j-th share of $\mathtt{share}(\boldsymbol{\tau}, \boldsymbol{\rho})$ is a linear combination of the values in $\boldsymbol{\tau}$ and $\boldsymbol{\rho}$. Thus, all parties can locally compute a degree-t Shamir sharing of the j-th share of $\mathtt{share}(\boldsymbol{\tau}, \boldsymbol{\rho})$ by using the degree-t Shamir sharings of the values in $\boldsymbol{\tau}$ and $\boldsymbol{\rho}$ prepared in Step 1 and applying linear combinations on their local shares. Let $[X_j]_t$ denote the resulting sharing.
3. For all $j \in \{1, 2, \ldots, n\}$, all parties send their shares of $[X_j]_t$ to P_j to let P_j reconstruct X_j. All parties take $\boldsymbol{X} = (X_1, \ldots, X_n)$ as output.

Note that $\boldsymbol{\tau}$ and $\boldsymbol{\rho}$ are all uniform field elements, and $\boldsymbol{X} = \mathtt{share}(\boldsymbol{\tau}, \boldsymbol{\rho})$. Therefore, the output $\boldsymbol{X}$ is a random Σ-sharing.

We note that this approach requires to prepare $\tilde{k}+\tilde{r} = O(n)$ random degree-t Shamir sharings and communicate n^2 field elements in order to prepare a random Σ-sharing, which is far from $O(n)$. To improve the efficiency, we try to prepare random sharings for a batch of (potentially different) secret sharing schemes each time.

10.1.2 Preparing Random Sharings for a Batch of Different Linear Secret Sharing Schemes

We note that the above vanilla process can be viewed as all parties securely evaluating a circuit for the sharing map $\mathtt{share}$ of Σ. In particular, (1) the circuit only involves linear operations, and (2) circuits for different secret sharing schemes (i.e., $\mathtt{share}_1, \mathtt{share}_2, \ldots, \mathtt{share}_k$) all satisfy that each output value is a linear combination of all input values with different coefficients. When we want to prepare random sharings for a batch of different secret sharing schemes, the joint circuit is very similar to a SIMD circuit (which is a circuit that contains many copies of the same sub-circuit). The only difference is that, in our case, each sub-circuit corresponds to a different secret sharing scheme, and therefore the coefficients used in different sub-circuits are distinct. On the other hand, a SIMD circuit would use the same coefficients in all sub-circuits. Thus, it motivates us to explore the packed secret-sharing technique in [34], which is originally used to evaluate a SIMD circuit.

Starting Idea. Suppose $\Sigma_1, \Sigma_2, \ldots, \Sigma_k$ are k arbitrary linear secret sharing schemes (Recall that we want to prepare random sharings for different sharing transformations, and every different sharing transformation requires to prepare a random sharing of a different secret sharing scheme). We assume that they all have share size 1 (i.e., each share is a single field element) for simplicity. We consider to use a packed secret sharing scheme that can store k secrets in each sharing. Our attempt is as follows:

1. All parties first prepare n random packed secret sharings (Our construction will use the packed Shamir secret sharings introduced below). The secrets are denoted by $\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_n$, where each secret $\boldsymbol{r}_j$ is a vector of k random elements in $\mathbb{F}$.
2. For all $i \in \{1, 2, \ldots, k\}$, we want to use the i-th values of all secret vectors to prepare a random sharing of Σ_i. With more details, suppose Σ_i has secret space $Z_i = \mathbb{F}^{\tilde{k}_i}$, and the sharing map of Σ_i is $\mathtt{share}_i : Z_i \times \mathbb{F}^{\tilde{r}_i} \to \mathbb{F}^n$. Consider the vector $(r_{1,i}, r_{2,i}, \ldots, r_{n,i})$ which contains the i-th values of all secret vectors. We plan to use the first $\tilde{k}_i$ values as the secret $\boldsymbol{\tau}_i$, and the next $\tilde{r}_i$ values as the random tape $\boldsymbol{\rho}_i$. Recall that we require $\mathtt{share}_i$ to be injective. We have $\tilde{k}_i + \tilde{r}_i \leq n$. Therefore, there are enough values for $\boldsymbol{\tau}_i$ and $\boldsymbol{\rho}_i$. The goal is to compute a random Σ_i-sharing $\boldsymbol{X}_i$ of the secret $\boldsymbol{\tau}_i$ with random tape $\boldsymbol{\rho}_i$, i.e., $\boldsymbol{X}_i = \mathtt{share}_i(\boldsymbol{\tau}_i, \boldsymbol{\rho}_i)$.
3. For each party P_j, let $\boldsymbol{u}_j$ denote the j-th shares of $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k$. We want to use the packed secret sharings of $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n$ to compute a single packed secret sharing of $\boldsymbol{u}_j$.
4. After obtaining a packed secret sharing of $\boldsymbol{u}_j$, we can reconstruct the sharing to P_j so that he learns the j-th share of each of $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k$. Thus, we start with n packed secret sharings (of $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_n$) of the same secret sharing scheme and end with k sharings $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k$ of k potentially different secret sharing schemes.

Clearly, the main question is how to realize Step 3. We observe that, since Σ_i is a linear secret sharing scheme, the j-th share of $\boldsymbol{X}_i$ can be written as a linear combination of the values in $\boldsymbol{\tau}_i$ and $\boldsymbol{\rho}_i$. Therefore, the j-th share of $\boldsymbol{X}_i$ is a linear combination of the values $(r_{1,i}, r_{2,i}, \ldots, r_{n,i})$. Since it holds for all $i \in \{1, 2, \ldots, k\}$, there exists constant vectors $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n \in \mathbb{F}^k$ such that

$$\boldsymbol{u}_j := \boldsymbol{c}_1 * \boldsymbol{r}_1 + \ldots + \boldsymbol{c}_n * \boldsymbol{r}_n,$$

where $*$ denotes the coordinate-wise multiplication operation. Thus, *what we need is a packed secret sharing scheme that supports efficient coordinate-wise multiplication with a constant vector.* We note that the packed Shamir secret sharing scheme fits our need as we show next.

Packed Shamir Secret Sharing Scheme and Multiplication-Friendliness. The packed Shamir secret sharing scheme [34] is a natural generalization of the standard Shamir secret sharing scheme [64]. It allows to secret-share a batch of secrets within a single Shamir sharing. For a vector $\boldsymbol{x} \in \mathbb{F}^k$, we use $[\boldsymbol{x}]_d$ to denote a degree-d packed Shamir sharing, where $k-1 \leq d \leq n-1$. It requires $d+1$ shares to reconstruct the whole sharing, and any $d-k+1$ shares are independent of the secrets. The packed Shamir secret sharing scheme has the following nice properties:

- Linear Homomorphism: For all $d \geq k-1$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\boldsymbol{x} + \boldsymbol{y}]_d = [\boldsymbol{x}]_d + [\boldsymbol{y}]_d$.
- Multiplicative: For all $d_1, d_2 \geq k-1$ subject to $d_1 + d_2 < n$, and for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\boldsymbol{x} * \boldsymbol{y}]_{d_1+d_2} = [\boldsymbol{x}]_{d_1} \cdot [\boldsymbol{y}]_{d_2}$, where the multiplications are performed on the corresponding

shares.

Note that when $d \leq n-k$, all parties can locally multiply a public vector $\boldsymbol{c} \in \mathbb{F}^k$ with a degree-d packed Shamir sharing $[\boldsymbol{x}]_d$:

1. All parties first locally compute a degree-$(k-1)$ packed Shamir sharing of $\boldsymbol{c}$, denoted by $[\boldsymbol{c}]_{k-1}$. Note that for a degree-$(k-1)$ packed Shamir sharing, all shares are determined by the secret $\boldsymbol{c}$.
2. All parties then locally compute $[\boldsymbol{c} * \boldsymbol{x}]_{n-1} = [\boldsymbol{c}]_{k-1} \cdot [\boldsymbol{x}]_{n-k}$.

We simply write $[\boldsymbol{c} * \boldsymbol{x}]_{n-1} = \boldsymbol{c} \cdot [\boldsymbol{x}]_{n-k}$ to denote the above process. We refer to this property as multiplication-friendliness.

To make sure that the packed Shamir secret sharing scheme is secure against t corrupted parties, we also require $d \geq t+k-1$. When $d = n-k$ and $k = (n-t+1)/2$, the degree-$(n-k)$ packed Shamir secret sharing scheme is both multiplication-friendly and secure against t corrupted parties.

Observe that when we use the degree-$(n-k)$ packed Shamir secret sharing scheme in our attempt, all parties can locally compute a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{u}_j$ by

$$[\boldsymbol{u}_j]_{n-1} = \boldsymbol{c}_1 \cdot [\boldsymbol{r}_1]_{n-k} + \ldots + \boldsymbol{c}_n \cdot [\boldsymbol{r}_n]_{n-k},$$

which solves the problem.

Summary of Our Construction. In summary, all parties run the following steps to prepare random sharings for k different linear secret sharing schemes $\Sigma_1, \Sigma_2, \ldots, \Sigma_k$.

1. Prepare Packed Shamir Sharings: All parties prepare n random degree-$(n-k)$ packed Shamir sharings, denoted by $[\boldsymbol{r}_1]_{n-k}, \ldots, [\boldsymbol{r}_n]_{n-k}$.
2. Use Packed Secrets as Randomness for Target LSSS: For all $i \in \{1, 2, \ldots, k\}$, let $\boldsymbol{\tau}_i = (r_{1,i}, \ldots, r_{\tilde{k}_i,i})$ and $\boldsymbol{\rho}_i = (r_{\tilde{k}_i+1,i}, \ldots, r_{\tilde{k}_i+\tilde{r}_i,i})$. Let $\boldsymbol{X}_i = \mathtt{share}_i(\boldsymbol{\tau}_i, \boldsymbol{\rho}_i)$.
3. Compute a Single Packed Shamir Sharing for All j-th Shares of Target LSSS via Local Operations: For all $j \in \{1, 2, \ldots, n\}$, let $\boldsymbol{u}_j$ be the j-th shares of $(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_k)$. All parties locally compute a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{u}_j$ by using $[\boldsymbol{r}_1]_{n-k}, \ldots, [\boldsymbol{r}_n]_{n-k}$. The resulting sharing is denoted by $[\boldsymbol{u}_j]_{n-1}$.
4. Reconstruct the Single Packed Shamir Sharing of All j-th Shares to P_j: For all $j \in \{1, 2, \ldots, n\}$, all parties reconstruct the sharing $[\boldsymbol{u}_j]_{n-1}$ to P_j to let him learn $\boldsymbol{u}_j = (u_j^{(1)}, \ldots, u_j^{(k)})$. Then all parties take $\{\boldsymbol{X}_i = (u_1^{(i)}, \ldots, u_n^{(i)})\}_{i=1}^k$ as output.

We note that in Step 4, $[\boldsymbol{u}_j]_{n-1}$ is not a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{u}_j$. *Directly sending the shares of $[\boldsymbol{u}_j]_{n-1}$ to P_j may leak the information about honest parties' shares.* To solve it, all parties also prepare n random degree-$(n-1)$ packed Shamir sharings of $\boldsymbol{0} \in \mathbb{F}^k$, denoted by $[\boldsymbol{o}_1]_{n-1}, \ldots, [\boldsymbol{o}_n]_{n-1}$. Then all parties use $[\boldsymbol{o}_j]_{n-1}$ to refresh the shares of $[\boldsymbol{u}_j]_{n-1}$ by computing $[\boldsymbol{u}_j]_{n-1} := [\boldsymbol{u}_j]_{n-1} + [\boldsymbol{o}_j]_{n-1}$. Now $[\boldsymbol{u}_j]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{u}_j$. All parties send their shares of $[\boldsymbol{u}_j]_{n-1}$ to P_j to let him reconstruct $\boldsymbol{u}_j$.

Communication Complexity. Thus, to prepare random sharings for k linear secret sharing schemes, our construction requires to prepare n random degree-$(n-k)$ packed Shamir sharings and n random degree-$(n-1)$ packed Shamir sharings of $\mathbf{0} \in \mathbb{F}^k$. And the communication complexity is n^2 field elements. On average, each random sharing costs $2n/k$ packed Shamir sharings and n^2/k elements of communication. When we use the generic approach to prepare random packed Shamir sharings, the total communication complexity per random sharing is $O(n^2/k)$ elements.

Recall that $k = (n-t+1)/2$. When $t = (1-\epsilon)\cdot n$ for a positive constant ϵ, the communication complexity per random sharing is $O(n)$ elements, which matches the communication complexity of delivering a random sharing by a trusted party up to a constant factor. In Chapter 11, we show that our technique works for any share size ℓ (with an increase in the communication complexity by a factor ℓ), and is naturally extended to any finite fields and rings $\mathbb{Z}/p^\ell\mathbb{Z}$.

Efficient Sharing Transformation. Recall that in the problem of sharing transformation, all parties start with holding a sharing $\boldsymbol{X}$ of a linear secret sharing scheme Σ. They want to compute a sharing $\boldsymbol{Y}$ of another linear secret sharing scheme Σ' such that the secret of $\boldsymbol{Y}$ is a linear map of the secret of $\boldsymbol{X}$.

As we discussed above, sharing transformation can be achieved efficiently with the help of a pair of random sharings $(\boldsymbol{R}, \boldsymbol{R}')$ such that $\boldsymbol{R}$ is a random Σ-sharing and $\boldsymbol{R}'$ is a random Σ'-sharing which satisfies that the secret of $\boldsymbol{R}'$ is equal to the result of applying the desired linear map on the secret of $\boldsymbol{R}$. *A key insight is that* $(\boldsymbol{R}, \boldsymbol{R}')$ *can just be seen as a linear secret sharing on its own.* With our technique of preparing random sharings for different linear secret sharing schemes, we can efficiently prepare a pair of random sharings $(\boldsymbol{R}, \boldsymbol{R}')$, allowing efficient sharing transformation from $\boldsymbol{X}$ to $\boldsymbol{Y}$.

When $t = (1-\epsilon)\cdot n$ for a positive constant ϵ, each sharing transformation only requires $O(n)$ field elements of communication.

10.2 Application: MPC via Packed Shamir Secret Sharing Schemes

In this section, we show that our technique for sharing transformation allows us to design an efficient MPC protocol via packed Shamir secret sharing schemes. We focus on the *dishonest majority setting* and information-theoretic setting in the circuit-independent preprocessing model. In the preprocessing model, all parties receive correlated randomness from a trusted party before the computation. The preprocessing model enables the possibility of an information-theoretic protocol in the dishonest majority setting, which otherwise cannot exist in the plain model. The cost of a protocol in the preprocessing model is measured by both the amount of preprocessing data prepared in the preprocessing phase and the amount of communication in the online phase [18, 25].

Let n be the number of parties, and t be the number of corrupted parties. For any positive constant ϵ, we show that there is an information-theoretic MPC protocol in the circuit-independent preprocessing model with semi-honest security (or malicious security) that computes an arith-

metic circuit C over a large finite field $\mathbb{F}$ (with $|\mathbb{F}| \geq |C| + n$) against $t = (1 - \epsilon) \cdot n$ corrupted parties with $O(|C|)$ field elements of preprocessing data and $O(|C|)$ field elements of communication.

Later on, we will propose an alternative solution for small finite fields of size $|\mathbb{F}| \geq 2n$ in Chapter 10.2.4.

Review the Packed Shamir Secret Sharing Scheme. We recall the notion of the packed Shamir secret sharing scheme. Let $\alpha_1, \ldots, \alpha_n$ be n distinct elements in $\mathbb{F}$ and $\mathtt{pos} = (p_1, p_2, \ldots, p_k)$ be another k distinct elements in $\mathbb{F}$. A *degree-d* ($d \geq k - 1$) packed Shamir sharing of $\boldsymbol{x} = (x_1, \ldots, x_k) \in \mathbb{F}^k$ is a vector $(w_1, \ldots, w_n)$ for which there exists a polynomial $f(\cdot) \in \mathbb{F}[X]$ of degree at most d such that $f(p_i) = x_i$ for all $i \in \{1, 2, \ldots, k\}$, and $f(\alpha_i) = w_i$ for all $i \in \{1, 2, \ldots, n\}$. The i-th share w_i is held by party P_i.

In our protocol, we will always use the same elements $\alpha_1, \ldots, \alpha_n$ for the positions of the shares of all parties. However, we may use different elements $\mathtt{pos}$ for the secrets. We will use $[\boldsymbol{x} \| \mathtt{pos}]_d$ to denote a degree-d packed Shamir sharing of $\boldsymbol{x} \in \mathbb{F}^k$ stored at positions $\mathtt{pos}$. Let $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_k)$ be distinct field elements in $\mathbb{F}$ that are different from $\alpha_1, \ldots, \alpha_n$. We will use $\boldsymbol{\beta}$ as the default positions for the secrets, and simply write $[\boldsymbol{x}]_d = [\boldsymbol{x} \| \boldsymbol{\beta}]_d$.

Recall that t is the number of corrupted parties. Let $k = (n - t + 1)/2$ and $d = n - k$. As we have shown in Chapter 10.1, all parties can locally multiply a public vector with a degree-$(n-k)$ packed Shamir sharing, and a degree-$(n-k)$ packed Shamir sharing is secure against t corrupted parties.

An Overview of Our Construction. At a high-level,

1. All parties start with sharing their input values by using packed Shamir sharings.
2. In each layer, addition gates and multiplication gates are divided into groups of size k. Each time we will evaluate a group of k gates:
   (a) For each group of k gates, all parties prepare two packed Shamir sharings, one for the first inputs of all gates, and the other one for the second inputs of all gates. Note that the secrets we want to be in a single sharing can be scattered in different output sharings from previous layers. This step is referred to as *network routing*.
   (b) After preparing the two input sharings, all parties evaluate these k gates. Addition gates can be locally computed since the packed Shamir secret sharing scheme is linearly homomorphic. For multiplication gates, we extend the technique of Beaver triples [5] to our setting, which we refer to as *packed Beaver triples*. All parties need to prepare packed Beaver triples in the preprocessing phase.
3. After evaluating the whole circuit, all parties reconstruct the sharings they hold to the parties who should receive the result.

Sparsely Packed Shamir Sharings. Our idea is to use a different position to store the output value of each gate. Recall that $|\mathbb{F}| \geq |C| + n$. Let $\beta_1, \beta_2, \ldots, \beta_{|C|}$ be $|C|$ distinct field elements that are different from $\alpha_1, \alpha_2, \ldots, \alpha_n$. (Recall that we have already defined $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_k)$, which are used as the default positions for a packed Shamir sharing.) We associate the field

element β_i with the i-th gate in C. We will use β_i as the position to store the output value of the i-th gate in a degree-$(n-k)$ packed Shamir sharing (see an example below).

Concretely, for each group of k gates, all parties will compute a degree-$(n-k)$ packed Shamir sharing such that the results are stored at the positions associated with these k gates respectively. For example, when $k = 3$, for a batch of 3 gates which are associated with the positions $\beta_1, \beta_3, \beta_6$ respectively, all parties will compute a degree-$(n-k)$ packed Shamir sharing $[(z_1, z_3, z_6)\|(\beta_1, \beta_3, \beta_6)]_{n-k}$ for this batch of gates, where z_1, z_3, z_6 are the output wires of these 3 gates.

10.2.1 Network Routing

In each intermediate layer, for every group of k gates, suppose $\boldsymbol{x}$ are the first inputs of these k gates, and $\boldsymbol{y}$ are the second inputs of these k gates. All parties will prepare two degree-$(n-k)$ packed Shamir sharings $[\boldsymbol{x}]_{n-k}$ and $[\boldsymbol{y}]_{n-k}$ stored at the default positions using the following approach. The reason of choosing the default positions is to use the packed Beaver triples, which use the default positions since the preprocessing phase is circuit-independent (discussed later). We focus on how to obtain $[\boldsymbol{x}]_{n-k}$.

Let $\boldsymbol{x} = (x_1, x_2, \ldots, x_k)$. For simplicity, we assume that $x_1, x_2, \ldots, x_k$ are output wires from k distinct gates. Later on, we will show how to handle the scenario where the same output wire is used multiple times by using fan-out operations. Since we use a different position to store the output of each gate, the positions of these k gates are all different. Let $p_1, \ldots, p_k$ denote the positions of these k gates and $\mathtt{pos} = (p_1, \ldots, p_k)$. We first show that all parties can locally compute a degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{x}\|\mathtt{pos}]_{n-1}$.

Selecting the Correct Secrets. For all $i \in \{1, 2, \ldots, k\}$, let $[\boldsymbol{x}^{(i)}\|\mathtt{pos}^{(i)}]_{n-k}$ be the degree-$(n-k)$ packed Shamir sharing that contains the secret x_i at position p_i from some previous layer. Let $\boldsymbol{e}_i$ be the i-th unit vector in $\mathbb{F}^k$ (i.e., only the i-th term is 1 and all other terms are 0). All parties locally compute a degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{e}_i\|\mathtt{pos}]_{k-1}$. Consider the following degree-$(n-1)$ packed Shamir sharing:

$$[\boldsymbol{e}_i\|\mathtt{pos}]_{k-1} \cdot [\boldsymbol{x}^{(i)}\|\mathtt{pos}^{(i)}]_{n-k}.$$

We claim that, the resulting sharing satisfies that the value stored at position p_i is x_i and the values stored at other positions in pos are all 0. To see this, recall that each packed Shamir sharing corresponds to a polynomial. Let f be the polynomial corresponding to $[\boldsymbol{e}_i\|\mathtt{pos}]_{k-1}$, and g be the polynomial corresponding to $[\boldsymbol{x}^{(i)}\|\mathtt{pos}^{(i)}]_{n-k}$. Then f satisfies that $f(p_i) = 1$ and $f(p_j) = 0$ for all $j \neq i$, and g satisfies that $g(p_i) = x_i$. Note that $h = f \cdot g$ is the polynomial corresponding to the resulting sharing $[\boldsymbol{e}_i\|\mathtt{pos}]_{k-1} \cdot [\boldsymbol{x}^{(i)}\|\mathtt{pos}^{(i)}]_{n-k}$, which satisfies that $h(p_i) = f(p_i) \cdot g(p_i) = 1 \cdot x_i = x_i$, and $h(p_j) = f(p_j) \cdot g(p_j) = 0 \cdot g(p_j) = 0$ for all $j \neq i$. Thus, the resulting sharing has value x_i in the position p_i and 0 in all other positions in pos. Effectively, we select the secret x_i from $[\boldsymbol{x}^{(i)}\|\mathtt{pos}^{(i)}]_{n-k}$ at position p_i and zero-out the values stored at other positions in pos.

Getting all Secrets into a Single Packed Shamir Sharing. Thus, for the following degree-$(n-1)$ packed Shamir sharing

$$\sum_{i=1}^{k} [\boldsymbol{e}_i \| \mathtt{pos}]_{k-1} \cdot [\boldsymbol{x}^{(i)} \| \mathtt{pos}^{(i)}]_{n-k},$$

it has value x_i stored in the position p_i for all $i \in \{1, 2, \ldots, k\}$, which means that it is a degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{x} \| \mathtt{pos}]_{n-1}$. Therefore, all parties can locally compute $[\boldsymbol{x} \| \mathtt{pos}]_{n-1} = \sum_{i=1}^{k} [\boldsymbol{e}_i \| \mathtt{pos}]_{k-1} \cdot [\boldsymbol{x}^{(i)} \| \mathtt{pos}^{(i)}]_{n-k}$.

Applying Sharing Transformation. Finally, to obtain $[\boldsymbol{x}]_{n-k} = [\boldsymbol{x} \| \boldsymbol{\beta}]_{n-k}$, all parties only need to do a sharing transformation from $[\boldsymbol{x} \| \mathtt{pos}]_{n-1}$ to $[\boldsymbol{x}]_{n-k}$. Relying on our technique for sharing transformation, we can achieve this step with $O(n)$ field elements of communication.

Therefore, our protocol for network routing only requires a local computation for $[\boldsymbol{x} \| \mathtt{pos}]_{n-1}$ and an efficient sharing transformation for $[\boldsymbol{x}]_{n-k}$ with $O(n)$ field elements of communication.

Handling Fan-out Operations. The above solution only works when all the wire values of $\boldsymbol{x}$ come from different gates. In a general case, $\boldsymbol{x}$ may contain many wire values from the same gate. We modify the above protocol as follows:

1. Suppose $x'_1, \ldots, x'_{k'}$ are the different values in $\boldsymbol{x}$. Let $\boldsymbol{x}' = (x'_1, \ldots, x'_{k'}, 0, \ldots, 0) \in \mathbb{F}^k$. For all $i \in \{1, 2, \ldots, k'\}$, let p_i be the position associated with the gate that outputs x'_i. We choose $p_{k'+1}, \ldots, p_k$ to be the first $(k-k')$ unused positions and set $\mathtt{pos} = (p_1, \ldots, p_k)$. Then, all parties follow a similar approach to locally compute a degree-$(n-1)$ packed Shamir sharing of $[\boldsymbol{x}' \| \mathtt{pos}]_{n-1}$.
2. Note that $\boldsymbol{x}'$ contains all different values in $\boldsymbol{x}$. Thus, there is a linear map $f : \mathbb{F}^k \to \mathbb{F}^k$ such that $\boldsymbol{x} = f(\boldsymbol{x}')$. Therefore, relying on our technique for sharing transformation, all parties transform $[\boldsymbol{x}' \| \mathtt{pos}]_{n-1}$ to $[\boldsymbol{x}]_{n-k}$.

The communication complexity remains $O(n)$ field elements.

10.2.2 Evaluating Multiplication Gates Using Packed Beaver Triples

For a group of k multiplication gates, suppose all parties have prepared two degree-$(n-k)$ packed Shamir sharings $[\boldsymbol{x}]_{n-k}$ and $[\boldsymbol{y}]_{n-k}$. Let $\mathtt{pos}$ be the positions associated with these k gates. The goal is to compute a degree-$(n-k)$ packed Shamir sharing of $\boldsymbol{x} * \boldsymbol{y}$ stored at positions $\mathtt{pos}$. To this end, we extend the technique of Beaver triples [5] to our setting, which we refer to as *packed Beaver triples*. We make use of a random packed Beaver triple $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-k})$, where $\boldsymbol{a}, \boldsymbol{b}$ are random vectors in $\mathbb{F}^k$ and $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. All parties run the following steps:

1. All parties locally compute $[\boldsymbol{x}+\boldsymbol{a}]_{n-k} = [\boldsymbol{x}]_{n-k} + [\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{n-k} = [\boldsymbol{y}]_{n-k} + [\boldsymbol{b}]_{n-k}$.
2. The first party P_1 collects the whole sharings $[\boldsymbol{x}+\boldsymbol{a}]_{n-k}, [\boldsymbol{y}+\boldsymbol{b}]_{n-k}$ and reconstructs the secrets $\boldsymbol{x}+\boldsymbol{a}, \boldsymbol{y}+\boldsymbol{b}$. Recall that $\boldsymbol{x} = (x_1, \ldots, x_k)$ and $\boldsymbol{a} = (a_1, \ldots, a_k)$ are vectors in $\mathbb{F}^k$, and $\boldsymbol{x}+\boldsymbol{a} = (x_1+a_1, \ldots, x_k+a_k)$. Similarly, $\boldsymbol{y}+\boldsymbol{b} = (y_1+b_1, \ldots, y_k+b_k)$. P_1 computes the sharings $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{k-1}$ and distributes the shares to other parties.

3. All parties locally compute

$$[\boldsymbol{z}]_{n-1} := [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{b}]_{n-k} - [\boldsymbol{y}+\boldsymbol{b}]_{k-1} \cdot [\boldsymbol{a}]_{n-k} + [\boldsymbol{c}]_{n-k}.$$

   Here the resulting sharing $[\boldsymbol{z}]_{n-1}$ has degree $n-1$ due to the second term and the third term.

4. Finally, all parties transform the sharing $[\boldsymbol{z}]_{n-1}$ to $[\boldsymbol{z}\|\texttt{pos}]_{n-k}$. Relying on our technique of sharing transformation, this can be done with $O(n)$ field elements of communication.

Note that in the above steps, all parties only reveal $[\boldsymbol{x}+\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{n-k}$ to P_1. Recall that $[\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{b}]_{n-k}$ are random degree-$(n-k)$ packed Shamir sharings. Therefore, $[\boldsymbol{x}+\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{n-k}$ are also random degree-$(n-k)$ packed Shamir sharings, which leak no information about $\boldsymbol{x}$ and $\boldsymbol{y}$ to P_1. Thus, the security follows.

Therefore, to evaluate a group of k multiplication gates, all parties need to prepare a random packed Beaver triple $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-k})$, which is of size $O(n)$ field elements. The communication complexity is $O(n)$ field elements.

10.2.3 Summary

In summary, our protocol works as follows. All parties first prepare enough packed Beaver triples stored at the default positions in the preprocessing phase. Then in the online phase, all parties evaluate the circuit layer by layer. For each layer, all parties first use the protocol for network routing to prepare degree-$(n-k)$ packed Shamir sharings for the inputs of this layer. Then, for every group of addition gates, all parties can compute them locally due to the linear homomorhpism of the packed Shamir secret sharing scheme. For every group of multiplication gates, we use the technique of packed Beaver triple to evaluate these gates. In particular, evaluating each group of multiplication gates will consume one fresh packed Beaver triple prepared in the preprocessing phase.

When $t = (1-\epsilon) \cdot n$ for a positive constant ϵ, we have $k = (n-t+1)/2 = O(n)$. For the amount of preprocessing data, we need to prepare a packed Beaver triple for each group of k multiplication gates. Thus, the amount of preprocessing data is bounded by $O(\frac{|C|}{k} \cdot n) = O(|C|)$. For the amount of communication, note that all parties need to communicate during the network routing and the evaluation of multiplication gates. Both protocols require $O(n)$ elements of communication to process k secrets. Thus, the amount of communication complexity is also bounded by $O(\frac{|C|}{k} \cdot n) = O(|C|)$.

Therefore, we obtain an information-theoretic MPC protocol in the circuit-independent preprocessing model with semi-honest security that computes an arithmetic circuit C over a large finite field $\mathbb{F}$ (with $|\mathbb{F}| \geq |C| + n$) against $t = (1-\epsilon) \cdot n$ corrupted parties with $O(|C|)$ field elements of preprocessing data and $O(|C|)$ field elements of communication.

10.2.4 An Alternative Approach for Network Routing

We note that the main reason of the use of a large finite field is due to the network routing: by storing the output value of each gate in a different position, all parties can locally collect the secrets they want and compute a single sharing for these secrets.

Now suppose we always use the default positions for the packed Shamir sharings. In this way, we only need the field size $|\mathbb{F}| \geq 2n$. However, an immediate problem is that *the secrets we need to be in a single sharing may come from the same positions*. On the other hand, if an input sharing satisfies that its secrets come from different positions in the output sharings of previous layers, we can prepare this input sharing as follows:

1. All parties first locally compute a degree-$(n-1)$ packed Shamir sharing such that it contains all the secrets we need but the order may be incorrect.
2. To obtain the input sharing we need, all parties perform a permutation on the secrets of the sharing that all parties have prepared in the first step.

To this end, we consider what we call the *non-collision* property stated in Property 10.1.

Property 10.1 (Non-collision)**.** *For each input sharing of each layer, the secrets of this input sharing come from different positions in the output sharings of previous layers.*

Unfortunately, this property does not hold in general. A counterexample is that we need the same secret twice in a single input sharing. Then these two secrets will always come from the same position. To solve this problem, we require that

- every output wire of the input layer and all intermediate layers is used exactly once as an input wire of a later layer (which may not be the next layer).

This requirement can be met by assuming that there is a fan-out gate right after each (input, addition, or multiplication) gate that copies the output wire the number of times it is used in later layers. We will discuss how to evaluate fan-out gates efficiently later. With this requirement, there is a bijective map between the output wires (of the input layer and all intermediate layers) and the input wires (of the output layer and all intermediate layers).

Note that only meeting this requirement is not enough: it is still possible that two secrets of a single input sharing come from the same position but in two different output sharings. Our idea is to perform a permutation on each output sharing to achieve the non-collision property.

Since every output wire from every layer is only used once as an input wire of another layer, the number of output sharings in the circuit is the same as the number of input sharings in the circuit. Let m denote the number of output packed Shamir sharings of the input layer and all intermediate layers in the circuit. Then the number of input packed Shamir sharings of the output layer and all intermediate layers is also m. We label all the output sharings by $1, 2, \ldots, m$ and all the input sharings also by $1, 2, \ldots, m$. Consider a matrix $\boldsymbol{N} \in \{1, 2, \ldots, m\}^{m \times k}$ where $\boldsymbol{N}_{i,j}$ is the index of the input sharing that the j-th secret of the i-th output sharing wants to go to. Then for all $\ell \in \{1, 2, \ldots, m\}$, there are exactly k entries of $\boldsymbol{N}$ which are equal to ℓ. We will prove the following theorem.

Theorem 10.1. *Let $m \geq 1, k \geq 1$ be integers. Let $\boldsymbol{N}$ be a matrix of dimension $m \times k$ in $\{1, 2, \ldots, m\}^{m \times k}$ such that for all $\ell \in \{1, 2, \ldots, m\}$, the number of entries of $\boldsymbol{N}$ which are equal to ℓ is k. Then, there exists m permutations $p_1, p_2, \ldots, p_m$ over $\{1, 2, \ldots, k\}$ such that after performing the permutation p_i on the i-th row of $\boldsymbol{N}$, the new matrix $\boldsymbol{N}'$ satisfies that each column of $\boldsymbol{N}'$ is a permutation over $(1, 2, \ldots, m)$. Furthermore, the permutations $p_1, p_2, \ldots, p_m$ can be found within polynomial time.*

Jumping ahead, when we apply p_i to the i-th output sharing for all $i \in \{1, 2, \ldots, m\}$, Theorem 10.1 guarantees that for all $j \in \{1, 2, \ldots, k\}$ the j-th secrets of all output sharings want to go to different input sharings. Note that this ensures the non-collision property. During the

computation, we will perform the permutation p_i on the i-th output sharing right after it is computed. Note that when preparing an input sharing, the secrets we need only come from the output sharings which have been computed. The secrets of these output sharings have been properly permuted such that the secrets we want are in different positions. With the non-collision property, we can achieve the network routing by following a similar approach to that in Chapter 10.2.1.

Evaluating Fan-Out Gates. We first model the problem as follows: given a degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{x}]_{n-k}$ along with a vector $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$, where $n_i \geq 1$ is the number of times that x_i is used in later layers, the goal is to compute $\frac{n_1+n_2+\ldots+n_k}{k}$ degree-$(n-k)$ packed Shamir sharings which contain n_i copies of the value x_i for all $i \in \{1, 2, \ldots, k\}$. (For simplicity, we assume that $n_1 + n_2 + \ldots + n_k$ is a multiple of k. We refer the readers to Chapter 13 for how we handle the edge case.) We first transform $\boldsymbol{x}$ to $m = \frac{n_1+n_2+\ldots+n_k}{k}$ vectors $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}$ in $\mathbb{F}^k$ such that they contain n_i copies of the value x_i for all $i \in \{1, 2, \ldots, k\}$. All parties use the following algorithm to locally determine what values should be in each of these m vectors.

1. All parties locally initiate an empty list L. From $i = 1$ to k, all parties locally insert n_i times of x_i into L.
2. From $i = 1$ to m, all parties locally set $\boldsymbol{x}^{(i)}$ to be the vector of the first k elements in L, and then remove these elements from L. In this way, all parties determine the values that should be in the m vectors $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}$.

For each $\boldsymbol{x}^{(i)} \in \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$, note that there is a linear map L'_i that maps $\boldsymbol{x}$ to $\boldsymbol{x}^{(i)}$. To obtain $[\boldsymbol{x}^{(i)}]_{n-k}$ from $[\boldsymbol{x}]_{n-k}$, all parties perform a sharing transformation to transform $[\boldsymbol{x}]_{n-k}$ to $[L'_i(\boldsymbol{x})]_{n-k}$.

Achieving Non-Collision Properties. We first show how to prove Theorem 10.1. Let $\boldsymbol{N}$ be the matrix in Theorem 10.1. Our idea is to repeat the following steps:

1. In the ℓ-th iteration, for each row of $\boldsymbol{N}$, we pick a value in the last $k - \ell + 1$ entries of this row (so that the first $\ell - 1$ entries will not be chosen), such that the values we pick in all rows form a permutation over $\{1, 2, \ldots, m\}$.
2. For each row of $\boldsymbol{N}$, we swap the ℓ-th entry with the value we picked in this row. In this way, the ℓ-th column of $\boldsymbol{N}$ is a permutation over $\{1, 2, \ldots, m\}$.

Note that in each iteration, we switch two elements in each row. At the end of the above process, we can compute the permutation for each row based on the elements we switched in each iteration.

To make this idea work, we need to show that we can always find the values which form a permutation over $\{1, 2, \ldots, m\}$ in Step 1. We note that this problem has a close connection to graph theory. We first introduce two basic notions.

- For a graph $G = (V, E)$, we say G is a bipartite graph if there exists a partition (V_1, V_2) of V such that all edges are between vertices in V_1 and vertices in V_2. Such a graph is denoted by $G = (V_1, V_2, E)$.
- For a bipartite graph $G = (V_1, V_2, E)$ where $|V_1| = |V_2|$, a perfect matching is a subset of edges $\mathcal{E} \in E$ which satisfies that each vertex in the sub-graph $(V_1, V_2, \mathcal{E})$ has degree

exactly 1.

Note that a permutation p over $\{1, 2, \ldots, m\}$ corresponds to a perfect matching in a bipartite graph: the set of vertices are $V_1 = V_2 = \{1, 2, \ldots, m\}$, and the set of edges are $\mathcal{E} = \{(i, p(i))\}_{i=1}^{m}$.

We transform this problem to finding a perfect matching in a bipartite graph. We explain our solution for the first iteration. Consider a graph $G = (V_1, V_2, E)$ where $V_1 = V_2 = \{1, 2, \ldots, m\}$. For each entry $N_{i,j}$, there is an edge $(i, N_{i,j})$ in E. Then picking a value in each row is equivalent to picking an edge for each vertex in V_1. The chosen values forming a permutation over $\{1, 2, \ldots, m\}$ is equivalent to the chosen edges forming a perfect matching in G.

We use Hall's Marriage Theorem to prove the existence of a perfect matching.

Theorem 10.2 (Hall's Marriage Theorem)**.** *For a bipartite graph (V_1, V_2, E) such that $|V_1| = |V_2|$, there exists a perfect matching iff for all subset $V_1' \subset V_1$, the number of the neighbors of vertices in V_2 is at least $|V_1'|$.*

Hall's Marriage Theorem is a well-known theorem in graph theory which has many applications in mathematics and computer science. It provides a necessary and sufficient condition of the existence of a perfect matching in a bipartite graph. In addition, there are known efficient polynomial-time algorithms to find a perfect matching in a bipartite graph, e.g. the Hopcroft-Karp algorithm.

To prove the existence of a perfect matching, we show that the graph G satisfies the necessary and sufficient condition in Hall's Marriage Theorem. We say a bipartite graph $G' = (V_1', V_2', E')$ is d-regular if the degree of each vertex in $V_1' \bigcup V_2'$ is d. A well-known corollary of Hall's Marriage Theorem states that:

Corollary 10.1. *There exists a perfect matching in a d-regular bipartite graph.*

Therefore, it is sufficient to show that the graph G is a d-regular bipartite graph.

For all vertex $i \in V_1$, there is an edge $(i, N_{i,j})$ in E for each entry in the i-th row of $\boldsymbol{N}$. Therefore, the degree of the vertex i is k. For all vertex $j \in V_2$, the degree of j equals to the number of entries in $\boldsymbol{N}$ which equal to j. Note that there are exactly k entries which equals to j. Thus, the degree of the vertex j is k. Therefore G is a k-regular graph. By Corollary 10.1, there exists a perfect matching in G. The same arguments work for other iterations. We refer the readers to Chapter 13 for more details.

With Theorem 10.1, the non-collision property can be achieved by performing a proper permutation on the secrets of each output sharing. This can be done by using our sharing transformation protocol.

Summary. Thus, the alternative approach for the network routing is as follows:

1. First, after evaluating a group of k gates in the current layer, all parties evaluate fan-out gates to copy each wire value the number of times that it will be used in the circuit.
2. Then, all parties perform a proper permutation on each output packed Shamir sharing of the current layer to achieve the non-collision property.
3. After achieving the non-collision property, for each input packed Shamir sharing of the next layer, all parties locally compute a packed Shamir sharing that contains the desired secrets. This is done in a similar way to the approach in Chapter 10.2.1.

4. Finally, they permute the secrets to the correct order to continue the computation.

We note that the first two steps can be done by using a single sharing transformation per input sharing since the composition of two linear maps is still a linear map.

10.2.5 Other Results

Malicious Security of the Online Protocol. To achieve malicious security, we extend the idea of using information-theoretic MACs introduced in [11, 30] to authenticate packed Shamir sharings. Concretely, at the beginning of the computation, all parties will prepare a random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{\gamma}]_{n-k}$, where $\boldsymbol{\gamma} = (\gamma, \gamma, \ldots, \gamma) \in \mathbb{F}^k$ and γ is a random field element. The secrets $\boldsymbol{\gamma}$ serve as the MAC key. To authenticate the secrets of a degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{x}]_{n-k}$, all parties will compute a degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{\gamma} * \boldsymbol{x}]_{n-k}$. We will show that almost all malicious behaviors of corrupted parties can be transformed to additive attacks, i.e., adding errors to the secrets of degree-$(n-k)$ packed Shamir sharings.

Note that if the corrupted parties change the secrets $\boldsymbol{x}$ to $\boldsymbol{x} + \boldsymbol{\delta}_1$, they also need to change the secrets $\boldsymbol{\gamma} * \boldsymbol{x}$ to $\boldsymbol{\gamma} * \boldsymbol{x} + \boldsymbol{\delta}_2$ such that $\boldsymbol{\delta}_2 = \boldsymbol{\gamma} * \boldsymbol{\delta}_1$. However, since γ is a uniform value in $\mathbb{F}$, the probability of a success attack is at most $1/|\mathbb{F}|$. When the field size is large enough, we can detect such an attack with overwhelming probability. See more details in Chapter 14.

Realizing Preprocessing in the Honest Majority Setting with Malicious Security. We then focus on the malicious security in the standard honest majority setting where the number of corrupted parties $t = (n-1)/2$. Our idea is to use an information-theoretic honest majority protocol to prepare the preprocessing data and then use our maliciously secure protocol to evaluate the circuit. Relying on our efficient multiplication protocol and efficient multiplication verification protocol constructed in Part I, we obtain an information-theoretic n-party MPC protocol which securely computes a single arithmetic circuit in the presence of a malicious adversary controlling up to $t = (n-1)/2$ parties with offline communication complexity $O(|C|n)$ and online communication complexity $O(|C|)$ among all parties. Note that the online communication is sublinear in the number of parties. To the best of our knowledge, this is the first work that achieves sublinear online communication complexity in the number of parties in the information-theoretic setting with honest majority. We refer the readers to Chapter 15.2 for more details.

Chapter 11

Preparing Random Sharings for Different Arithmetic Secret Sharing Schemes

11.1 Arithmetic Secret Sharing Schemes

Let $\mathcal{R}$ be a finite commutative ring. In this work, we consider the following arithmetic secret sharing schemes from [2] (with slight modifications).

Definition 11.1 (Arithmetic Secret Sharing Schemes)**.** *The syntax of an $\mathcal{R}$-arithmetic secret sharing scheme Σ consists of the following data:*

- *A set of parties $\mathcal{I} = \{1, \ldots, n\}$.*
- *A secret space $Z = \mathcal{R}^k$. k is also denoted as the number of secrets packed within Σ.*
- *A share space $U = \mathcal{R}^\ell$. ℓ is also denoted as the share size.*
- *A sharing space $C \subset U^{\mathcal{I}}$, where $U^{\mathcal{I}}$ denotes the indexed Cartesian product $\prod_{i\in\mathcal{I}} U$.*
- *An injective $\mathcal{R}$-module homomorphism: $\mathtt{share} : Z \times \mathcal{R}^r \to C$, which maps a secret $\boldsymbol{x} \in Z$ and a random tape $\boldsymbol{\rho} \in \mathcal{R}^r$, to a sharing $\boldsymbol{X} \in C$. $\mathtt{share}$ is also denoted as the sharing map of Σ.*
- *A surjective $\mathcal{R}$-module homomorphism: $\mathtt{rec} : C \to Z$, which takes as input a sharing $\boldsymbol{X} \in C$ and outputs a secret $\boldsymbol{x} \in Z$. $\mathtt{rec}$ is also denoted as the reconstruction map of Σ.*

The scheme Σ satisfies that for all $\boldsymbol{x} \in Z$ and $\boldsymbol{\rho} \in \mathcal{R}^r$, $\mathtt{rec}(\mathtt{share}(\boldsymbol{x}, \boldsymbol{\rho})) = \boldsymbol{x}$. We may refer to Σ as the 6-tuple $(n, Z, U, C, \mathtt{share}, \mathtt{rec})$.

For a non-empty set $A \subset \mathcal{I}$, the natural projection π_A maps a tuple $u = (u_i)_{i\in\mathcal{I}} \in U^{\mathcal{I}}$ to the tuple $(u_i)_{i\in A} \in U^A$.

Definition 11.2 (Privacy Set and Reconstruction Set)**.** *Suppose $A \subset \mathcal{I}$ is nonempty. We say A is a privacy set if for all $\boldsymbol{x}_0, \boldsymbol{x}_1 \in Z$, and for all vector $\boldsymbol{v} \in U^A$,*

$$\Pr_{\boldsymbol{\rho}}[\pi_A(\mathtt{share}(\boldsymbol{x}_0, \boldsymbol{\rho})) = \boldsymbol{v}] \;=\; \Pr_{\boldsymbol{\rho}}[\pi_A(\mathtt{share}(\boldsymbol{x}_1, \boldsymbol{\rho})) = \boldsymbol{v}].$$

We say A is a reconstruction set if there is an $\mathcal{R}$-module homomorphism $\mathtt{rec}_A : \pi_A(C) \to Z$, such that for all $\boldsymbol{X} \in C$,

$$\mathtt{rec}_A(\pi_A(\boldsymbol{X})) = \mathtt{rec}(\boldsymbol{X}).$$

Intuitively, for a privacy set A, the shares of parties in A are independent of the secret. For a reconstruction set A, the shares of parties in A fully determine the secret.

Threshold Linear Secret Sharing Schemes and Multiplication-friendly Property. In this work, we are interested in threshold arithmetic secret sharing schemes. Concretely, for a positive integer $t < n$, a threshold-t arithmetic secret sharing scheme satisfies that for all $A \subset \mathcal{I}$ with $|A| \leq t$, A is a privacy set.

We are interested in the following property.

Property 11.1 (Multiplication-Friendliness)**.** *We say $\Sigma = (n, Z = \mathcal{R}^k, U, C, \mathtt{share}, \mathtt{rec})$ is multiplication-friendly if there is an $\mathcal{R}$-arithmetic secret sharing scheme $\Sigma' = (n, Z = \mathcal{R}^k, U', C', \mathtt{share}', \mathtt{rec}')$ and n functions $\{f_i : \mathcal{R}^k \times U \to U'\}_{i=1}^n$ such that for all $\boldsymbol{c} \in \mathcal{R}^k$ and for all $\boldsymbol{X} \in C$,*

- *$\boldsymbol{Y} = (f_1(\boldsymbol{c}, X_1), f_2(\boldsymbol{c}, X_2), \ldots, f_n(\boldsymbol{c}, X_n))$ is in C', i.e., a sharing in Σ'. We will use $\boldsymbol{Y} = \boldsymbol{c} \cdot \boldsymbol{X}$ to represent the computation process from $\boldsymbol{c}$ and $\boldsymbol{X}$ to $\boldsymbol{Y}$.*
- *$\mathtt{rec}'(\boldsymbol{Y}) = \boldsymbol{c} * \mathtt{rec}(\boldsymbol{X})$, where $*$ is the coordinate-wise multiplication operation.*

Intuitively, for a multiplication-friendly scheme Σ, if all parties hold a Σ-sharing of a secret $\boldsymbol{x} \in Z$ and a public vector $\boldsymbol{c} \in \mathcal{R}^k$, they can locally compute a Σ'-sharing of the secret $\boldsymbol{c} * \boldsymbol{x}$, where $*$ denotes the coordinate-wise multiplication operation. We have the following lemma.

Lemma 11.1. *If Σ is a multiplication-friendly threshold-t $\mathcal{R}$-arithmetic secret sharing scheme, and Σ' be the $\mathcal{R}$-arithmetic secret sharing scheme defined in Property 11.1, then Σ' has threshold t.*

Proof. We will prove that for all set A of t parties, for all $\boldsymbol{x}_0, \boldsymbol{x}_1 \in Z$, and for all vector $\boldsymbol{v}' \in (U')^A$,

$$\Pr_{\boldsymbol{\rho}'}[\pi_A(\mathtt{share}'(\boldsymbol{x}_0, \boldsymbol{\rho}')) = \boldsymbol{v}'] \;=\; \Pr_{\boldsymbol{\rho}'}[\pi_A(\mathtt{share}'(\boldsymbol{x}_1, \boldsymbol{\rho}')) = \boldsymbol{v}'].$$

Since Σ has threshold t, for all $\boldsymbol{v} \in U^A$, we have

$$\Pr_{\boldsymbol{\rho}}[\pi_A(\mathtt{share}(\boldsymbol{x}_0, \boldsymbol{\rho})) = \boldsymbol{v}] \;=\; \Pr_{\boldsymbol{\rho}}[\pi_A(\mathtt{share}(\boldsymbol{x}_1, \boldsymbol{\rho})) = \boldsymbol{v}].$$

Let $\mathbf{1}$ denote the vector $(1, 1, \ldots, 1) \in Z$ and $\mathbf{0}$ denote the vector $(0, 0, \ldots, 0) \in Z$. Then, for all $\boldsymbol{v}' \in (U')^A$,

$$\Pr_{\boldsymbol{\rho}, \boldsymbol{\rho}'}[\pi_A(\mathbf{1} \cdot \mathtt{share}(\boldsymbol{x}_0, \boldsymbol{\rho}) + \mathtt{share}'(\mathbf{0}, \boldsymbol{\rho}')) = \boldsymbol{v}'] \;=\; \Pr_{\boldsymbol{\rho}, \boldsymbol{\rho}'}[\pi_A(\mathbf{1} \cdot \mathtt{share}(\boldsymbol{x}_1, \boldsymbol{\rho}) + \mathtt{share}'(\mathbf{0}, \boldsymbol{\rho}')) = \boldsymbol{v}'].$$

It is sufficient to show that, for all $\boldsymbol{x} \in Z$, $\mathbf{1} \cdot \mathtt{share}(\boldsymbol{x}, \boldsymbol{\rho}) + \mathtt{share}'(\mathbf{0}, \boldsymbol{\rho}')$ is a random Σ'-sharing of $\boldsymbol{x}$. Note that, $\mathbf{1} \cdot \mathtt{share}(\boldsymbol{x}, \boldsymbol{\rho})$ is a Σ'-sharing of $\boldsymbol{x}$. Then there exists $\boldsymbol{\rho}''$ such that $\mathtt{share}'(\boldsymbol{x}, \boldsymbol{\rho}'') = \mathbf{1} \cdot \mathtt{share}(\boldsymbol{x}, \boldsymbol{\rho})$. Thus $\mathbf{1} \cdot \mathtt{share}(\boldsymbol{x}, \boldsymbol{\rho}) + \mathtt{share}'(\mathbf{0}, \boldsymbol{\rho}') = \mathtt{share}'(\boldsymbol{x}, \boldsymbol{\rho}'') + \mathtt{share}'(\mathbf{0}, \boldsymbol{\rho}') = \mathtt{share}'(\boldsymbol{x}, \boldsymbol{\rho}' + \boldsymbol{\rho}'')$. The last step follows from the fact that $\mathtt{share}'$ is an $\mathcal{R}$-module homomorphism. When $\boldsymbol{\rho}'$ is uniformly random, $\boldsymbol{\rho}' + \boldsymbol{\rho}''$ is also uniformly random. Thus $\mathtt{share}'(\boldsymbol{x}, \boldsymbol{\rho}' + \boldsymbol{\rho}'')$ is a random Σ'-sharing of $\boldsymbol{x}$. □

11.2 Packed Shamir Secret Sharing Scheme

In our work, we are interested in the packed Shamir secret sharing scheme. We use the packed secret-sharing technique introduced by Franklin and Yung [34]. This is a generalization of the standard Shamir secret sharing scheme [64]. Let $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq 2n$. Let n be the number of parties and k be the number of secrets that are packed in one sharing. Let $\alpha_1, \ldots, \alpha_n$ be n distinct elements in $\mathbb{F}$ and $\texttt{pos} = (p_1, p_2, \ldots, p_k)$ be another k distinct elements in $\mathbb{F}$. A *degree-d* ($d \geq k-1$) packed Shamir sharing of $\boldsymbol{x} = (x_1, \ldots, x_k) \in \mathbb{F}^k$ is a vector $(w_1, \ldots, w_n)$ for which there exists a polynomial $f(\cdot) \in \mathbb{F}[X]$ of degree at most d such that $f(p_i) = x_i$ for all $i \in \{1, 2, \ldots, k\}$, and $f(\alpha_i) = w_i$ for all $i \in \{1, 2, \ldots, n\}$. The i-th share w_i is held by party P_i. Reconstructing a degree-d packed Shamir sharing requires $d+1$ shares and can be done by Lagrange interpolation. For a random degree-d packed Shamir sharing of $\boldsymbol{x}$, any $d-k+1$ shares are independent of the secret $\boldsymbol{x}$.

In our work, we will always use the same elements $\alpha_1, \ldots, \alpha_n$ for the shares of all parties. However, we may use different elements $\texttt{pos}$ for the secrets. We will use $[\boldsymbol{x}\|\texttt{pos}]_d$ to denote a degree-d packed Shamir sharing of $\boldsymbol{x} \in \mathbb{F}^k$ stored at positions $\texttt{pos}$. In the following, operations (addition and multiplication) between two packed Shamir sharings are coordinate-wise. We recall two properties of the packed Shamir sharing scheme:

- Linear Homomorphism: For all $d \geq k-1$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\boldsymbol{x}+\boldsymbol{y}\|\texttt{pos}]_d = [\boldsymbol{x}\|\texttt{pos}]_d + [\boldsymbol{y}\|\texttt{pos}]_d$.
- Multiplicative: Let $*$ denote the coordinate-wise multiplication operation. For all $d_1, d_2 \geq k-1$ subject to $d_1+d_2 < n$, and for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\boldsymbol{x}*\boldsymbol{y}\|\texttt{pos}]_{d_1+d_2} = [\boldsymbol{x}\|\texttt{pos}]_{d_1} \cdot [\boldsymbol{y}\|\texttt{pos}]_{d_2}$.

These two properties directly follow from the computation of the underlying polynomials.

Note that the second property implies that, for all $k-1 \leq d \leq n-k$, a degree-d packed Shamir secret sharing scheme is multiplication-friendly (defined in Property 11.1). Concretely, for all $\boldsymbol{x}, \boldsymbol{c} \in \mathbb{F}^k$, all parties can locally compute $[\boldsymbol{c} * \boldsymbol{x}\|\texttt{pos}]_{d+k-1}$ from $[\boldsymbol{x}\|\texttt{pos}]_d$ and the public vector $\boldsymbol{c}$. To see this, all parties can locally transform $\boldsymbol{c}$ to a degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{c}\|\texttt{pos}]_{k-1}$. Then, they can use the property of the packed Shamir sharing scheme to compute $[\boldsymbol{c} * \boldsymbol{x}\|\texttt{pos}]_{d+k-1} = [\boldsymbol{c}\|\texttt{pos}]_{k-1} \cdot [\boldsymbol{x}\|\texttt{pos}]_d$.

Recall that t is the number of corrupted parties. Also recall that a degree-d packed Shamir secret sharing scheme is of threshold $d-k+1$. To ensure that the packed Shamir secret sharing scheme has threshold t and is multiplication-friendly, we choose k such that $t \leq d-k+1$ and $d \leq n-k$. When $d = n-k$ and $k = (n-t+1)/2$, both requirements hold and k is maximal.

11.3 Preparing Random Sharings for Different Arithmetic Secret Sharing Schemes

In this part, we introduce our main contribution: an efficient protocol that prepares random sharings for a batch of different arithmetic secret sharing schemes. Let $\mathcal{R}$ be a finite commutative ring. Let $\Pi = (n, \tilde{Z}, \tilde{U}, \tilde{C}, \texttt{share}_\Pi, \texttt{rec}_\Pi)$ be an $\mathcal{R}$-arithmetic secret sharing scheme. Our goal is to realize the functionality $\mathcal{F}_{\text{rand-sharing}}$ presented in Functionality 11.1.

Figure 11.1: Functionality $\mathcal{F}_{\text{rand-sharing}}(\Pi)$

1. $\mathcal{F}_{\text{rand-sharing}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$.
2. $\mathcal{F}_{\text{rand-sharing}}$ receives from the adversary a set of shares $\{\boldsymbol{u}_j\}_{j\in\mathcal{C}orr}$ where $\boldsymbol{u}_j \in \tilde{U}$ for all $j \in \mathcal{C}orr$.
3. $\mathcal{F}_{\text{rand-sharing}}$ samples a random Π-sharing $\boldsymbol{X}$ such that the shares of $\boldsymbol{X}$ held by corrupted parties are identical to those received from the adversary, i.e., $\pi_{\mathcal{C}orr}(\boldsymbol{X}) = (\boldsymbol{u}_j)_{j\in\mathcal{C}orr}$. If such a sharing does not exist, $\mathcal{F}_{\text{rand-sharing}}$ sends `abort` to all honest parties and halts.
4. Otherwise, $\mathcal{F}_{\text{rand-sharing}}$ distributes the shares of $\boldsymbol{X}$ to honest parties.

Initialization. Let $\Sigma = (n, Z = \mathcal{R}^k, U, C, \texttt{share}, \texttt{rec})$ be a multiplication-friendly threshold-t $\mathcal{R}$-arithmetic secret sharing scheme. In the following, we will use $[\boldsymbol{x}]$ to denote a Σ-sharing of $\boldsymbol{x} \in \mathcal{R}^k$. Let $\Sigma' = (n, Z' = \mathcal{R}^k, U', C', \texttt{share}', \texttt{rec}')$ be the $\mathcal{R}$-arithmetic secret sharing scheme in Property 11.1. By Lemma 11.1, Σ' has threshold t. We use $\langle \boldsymbol{y} \rangle$ to denote a Σ'-sharing of $\boldsymbol{y} \in \mathcal{R}^k$. For all $\boldsymbol{c} \in \mathcal{R}^k$, we will write

$$\langle \boldsymbol{c} * \boldsymbol{x} \rangle = \boldsymbol{c} \cdot [\boldsymbol{x}]$$

to represent the computation process from $\boldsymbol{c}$ and $[\boldsymbol{x}]$ to $\langle \boldsymbol{c} * \boldsymbol{x} \rangle$ in Property 11.1.

Our construction will use the ideal functionality $\mathcal{F}_{\text{rand}} = \mathcal{F}_{\text{rand-sharing}}(\Sigma)$ that prepares a random Σ-sharing, and the ideal functionality $\mathcal{F}_{\text{randZero}}$ (Functionality 11.2) that prepares a random Σ'-sharing of $\mathbf{0} \in \mathcal{R}^k$.

Figure 11.2: Functionality $\mathcal{F}_{\text{randZero}}$

1. Let $\Sigma' = (n, Z' = \mathcal{R}^k, U', C', \texttt{share}', \texttt{rec}')$. $\mathcal{F}_{\text{randZero}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$.
2. $\mathcal{F}_{\text{randZero}}$ receives from the adversary a set of shares $\{\boldsymbol{u}'_j\}_{j\in\mathcal{C}orr}$, where $\boldsymbol{u}'_j \in U'$ for all $P_j \in \mathcal{C}orr$.
3. $\mathcal{F}_{\text{randZero}}$ samples a random Σ'-sharing of $\mathbf{0} \in \mathcal{R}^k$, $\langle \mathbf{0} \rangle$, such that the shares of corrupted parties are identical to those received from the adversary, i.e., $\pi_{\mathcal{C}orr}(\langle \mathbf{0} \rangle) = (\boldsymbol{u}'_j)_{j\in\mathcal{C}orr}$. If such a sharing does not exist, $\mathcal{F}_{\text{randZero}}$ sends `abort` to all honest parties and halts.
4. Otherwise, $\mathcal{F}_{\text{randZero}}$ distributes the shares of $\langle \mathbf{0} \rangle$ to honest parties.

Let $\Pi_1, \Pi_2, \ldots, \Pi_k$ be k arbitrary $\mathcal{R}$-arithmetic secret sharing schemes with the restriction that all schemes have the same share size, i.e., the share space $\tilde{U} = \mathcal{R}^{\tilde{\ell}}$. Let $\tilde{Z}_i = \mathcal{R}^{\tilde{k}_i}$ be the secret space of Π_i and $\texttt{share}_i : \tilde{Z}_i \times \mathcal{R}^{\tilde{r}_i} \to \tilde{C}_i$ be the sharing map. Since $\texttt{share}_i$ is injective, and $\tilde{C}_i \subset \tilde{U}^{\mathcal{I}}$, we have $\tilde{k}_i + \tilde{r}_i \leq n \cdot \tilde{\ell}$.

The goal is to prepare k random sharings $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_k$ such that $\boldsymbol{X}_i$ is a random Π_i-sharing, i.e., realizing $\{\mathcal{F}_{\text{rand-sharing}}(\Pi_i)\}_{i=1}^k$.

Protocol Description. The construction of our protocol RAND-SHARING appears in Protocol 11.3.

Lemma 11.2. *For any k $\mathcal{R}$-arithmetic secret sharing schemes $\{\Pi_i\}_{i=1}^k$ such that they have the same share size, Protocol* RAND-SHARING *securely computes $\{\mathcal{F}_{\text{rand-sharing}}(\Pi_i)\}_{i=1}^k$ in the $\{\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{randZero}}\}$-hybrid model against a semi-honest adversary who controls t parties.*

Proof. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}orr$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties.

The simulator $\mathcal{S}$ works as follows.

1. In Step 2, $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{rand}}$: For all $v \in \{1, 2, \ldots, n \cdot \tilde{\ell}\}$, $\mathcal{S}$ receives the shares of $[\boldsymbol{r}_v]$ held by corrupted parties.
2. In Step 3, $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{randZero}}$: For all $j \in \{1, 2, \ldots, n\}$ and $m \in \{1, 2, \ldots, \tilde{\ell}\}$, $\mathcal{S}$ receives the shares of $\langle \boldsymbol{o}_j^{(m)} \rangle$ held by corrupted parties.
3. In Step 4, for all $j \in \{1, 2, \ldots, n\}$, $m \in \{1, 2, \ldots, \tilde{\ell}\}$, and $v \in \{1, \ldots, n \cdot \tilde{\ell}\}$, $\mathcal{S}$ locally compute $\boldsymbol{c}_{j,v}^{(\star,m)}$ by following the protocol.
4. In Step 5, for all $j \in \{1, 2, \ldots, n\}$ and $m \in \{1, 2, \ldots, \tilde{\ell}\}$, $\mathcal{S}$ follows the protocol and computes the shares of $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ held by corrupted parties. $\mathcal{S}$ simulates the shares that are sent from honest parties to corrupted parties as follows:
   (a) For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{S}$ samples a random Π_i-sharing and obtains the shares of corrupted parties $\{\boldsymbol{u}_j^{(i)}\}_{j \in \mathcal{C}orr}$, where $\boldsymbol{u}_j^{(i)} = (u_j^{(i,1)}, u_j^{(i,2)}, \ldots, u_j^{(i,\tilde{\ell})}) \in \mathcal{R}^{\tilde{\ell}}$.
   (b) Then for all $j \in \{1, 2, \ldots, n\}$ and $m \in \{1, 2, \ldots, \tilde{\ell}\}$, $\mathcal{S}$ computes $\boldsymbol{u}_j^{(\star,m)}$.
   (c) According to Lemma 11.1, Σ' has threshold t. Therefore, the shares of a random Σ'-sharing are independent of its secret. For all $j \in \mathcal{C}orr$ and $m \in \{1, 2, \ldots, \tilde{\ell}\}$, $\mathcal{S}$ samples a random Σ'-sharing $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ based on the secret $\boldsymbol{u}_j^{(\star,m)}$ and the shares of $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ held by corrupted parties computed by $\mathcal{S}$.
   (d) Finally, $\mathcal{S}$ sends the shares of $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ held by honest parties to the adversary.
5. In Step 6, for all $i \in \{1, 2, \ldots, k\}$, $\mathcal{S}$ sends the shares $\{\boldsymbol{u}_j^{(i)}\}_{j \in \mathcal{C}orr}$ to $\mathcal{F}_{\text{rand-sharing}}(\Pi_i)$.

This completes the description of the simulator. We show that the simulator $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Note that all parties only communicate with other parties in Step 5.

- We first show that the shares of corrupted parties, $\{\boldsymbol{u}_j^{(i)}\}_{j \in \mathcal{C}orr}$, sampled by $\mathcal{S}$ have the same distribution as those in the real world. In the ideal world, $\mathcal{S}$ randomly samples the shares of corrupted parties by first sampling random sharings of $\Pi_1, \Pi_2, \ldots, \Pi_k$ and then obtaining the shares of corrupted parties. In the real world, the shares of corrupted parties are computed by following $\texttt{share}_i(\boldsymbol{\tau}_i, \boldsymbol{\rho}_i)$ for all $i \in \{1, 2, \ldots, k\}$. Recall that $\boldsymbol{\tau}_i, \boldsymbol{\rho}_i$ are generated by $\mathcal{F}_{\text{rand}}$. And Σ has threshold t. Therefore, $\boldsymbol{\tau}_i$ and $\boldsymbol{\rho}_i$ are uniformly random. Thus, $\{\boldsymbol{u}_j^{(i)}\}_{j \in \mathcal{C}orr}$ have the same distribution in both the ideal world and the real world.
- Then, we show that for all $j \in \mathcal{C}orr$ and $m \in \{1, 2, \ldots, \tilde{\ell}\}$, the shares of $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ held by honest parties (which are supposed to send to corrupted parties) generated by $\mathcal{S}$ have the same distribution as those in the real world. Recall that $\langle \boldsymbol{o}_j^{(m)} \rangle$ is a random Σ'-sharing

Figure 11.3: Protocol RAND-SHARING

1. Let $\Pi_1, \ldots, \Pi_k$ be k arbitrary $\mathcal{R}$-arithmetic secret sharing schemes with the same share space $\tilde{U} = \mathcal{R}^{\tilde{\ell}}$. For all $i \in \{1, \ldots, k\}$, let $\tilde{Z}_i = \mathcal{R}^{\tilde{k}_i}$ be the secret space of Π_i, and $\mathtt{share}_i : \tilde{Z}_i \times \mathcal{R}^{\tilde{r}_i} \to \tilde{C}_i$ be the sharing map of Π_i. We have $\tilde{k}_i + \tilde{r}_i \leq n \cdot \tilde{\ell}$.
2. All parties invoke $\mathcal{F}_{\text{rand}}$ and obtain $n \cdot \tilde{\ell}$ random Σ-sharings, $[\boldsymbol{r}_1], \ldots, [\boldsymbol{r}_{n\cdot\tilde{\ell}}]$. For all $i \in \{1, \ldots, k\}$, let $\boldsymbol{\tau}_i = (r_{1,i}, \ldots, r_{\tilde{k}_i,i}) \in \mathcal{R}^{\tilde{k}_i}$, and $\boldsymbol{\rho}_i = (r_{\tilde{k}_i+1,i}, r_{\tilde{k}_i+2,i}, \ldots, r_{\tilde{k}_i+\tilde{r}_i,i}) \in \mathcal{R}^{\tilde{r}_i}$. The goal is to compute the Π_i-sharing $\boldsymbol{X}_i = \mathtt{share}_i(\boldsymbol{\tau}_i, \boldsymbol{\rho}_i)$.
3. All parties invoke $\mathcal{F}_{\text{randZero}}$ $n \cdot \tilde{\ell}$ times and obtain $n \cdot \tilde{\ell}$ random Σ'-sharings of $\mathbf{0} \in \mathcal{R}^k$, denoted by $\{\langle \boldsymbol{o}_j^{(1)} \rangle, \langle \boldsymbol{o}_j^{(2)} \rangle, \ldots, \langle \boldsymbol{o}_j^{(\tilde{\ell})} \rangle\}_{j=1}^n$.
4. For all $i \in \{1, 2, \ldots, k\}$, $j \in \{1, 2, \ldots, n\}$, and $m \in \{1, 2, \ldots, \tilde{\ell}\}$, let $\mathcal{L}_j^{(i,m)} : \tilde{Z}_i \times \mathcal{R}^{\tilde{r}_i} \to \mathcal{R}$ denote the $\mathcal{R}$-module homomorphism such that for all $\boldsymbol{\tau} \in \tilde{Z}_i$ and $\boldsymbol{\rho} \in \mathcal{R}^{\tilde{r}_i}$, $\mathcal{L}_j^{(i,m)}(\boldsymbol{\tau}, \boldsymbol{\rho})$ outputs the m-th element of the j-th share of the Π_i-sharing $\mathtt{share}_i(\boldsymbol{\tau}, \boldsymbol{\rho})$. Then there exist $c_{j,1}^{(i,m)}, \ldots, c_{j,\tilde{k}_i+\tilde{r}_i}^{(i,m)} \in \mathcal{R}$ such that

$$\mathcal{L}_j^{(i,m)}(\boldsymbol{\tau}, \boldsymbol{\rho}) = \sum_{v=1}^{\tilde{k}_i} c_{j,v}^{(i,m)} \cdot \tau_v + \sum_{v=1}^{\tilde{r}_i} c_{j,\tilde{k}_i+v}^{(i,m)} \cdot \rho_v.$$

   For all $j \in \{1, 2, \ldots, n\}$, $m \in \{1, 2, \ldots, \tilde{\ell}\}$, and $v \in \{1, \ldots, n \cdot \tilde{\ell}\}$, let

$$\boldsymbol{c}_{j,v}^{(\star,m)} = (c_{j,v}^{(1,m)}, c_{j,v}^{(2,m)}, \ldots, c_{j,v}^{(k,m)}) \in \mathcal{R}^k,$$

   where $c_{j,v}^{(i,m)} = 0$ for all $v > \tilde{k}_i + \tilde{r}_i$.
5. For all $i \in \{1, 2, \ldots, k\}$, $j \in \{1, 2, \ldots, n\}$, and $m \in \{1, 2, \ldots, \tilde{\ell}\}$, let $u_j^{(i,m)} = \mathcal{L}_j^{(i,m)}(\boldsymbol{\tau}_i, \boldsymbol{\rho}_i)$. Let $\boldsymbol{u}_j^{(\star,m)} = (u_j^{(1,m)}, u_j^{(2,m)}, \ldots, u_j^{(k,m)})$. For all $j \in \{1, 2, \ldots, n\}$ and $m \in \{1, 2, \ldots, \tilde{\ell}\}$, all parties locally compute a Σ'-sharing

$$\langle \boldsymbol{u}_j^{(\star,m)} \rangle = \langle \boldsymbol{o}_j^{(m)} \rangle + \sum_{v=1}^{n\cdot\tilde{\ell}} \boldsymbol{c}_{j,v}^{(\star,m)} \cdot [\boldsymbol{r}_v].$$

   Then, all parties send their shares of $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ to P_j.
6. For all $j \in \{1, 2, \ldots, n\}$ and $m \in \{1, 2, \ldots, \tilde{\ell}\}$, P_j reconstructs the Σ'-sharing $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ and learns $\boldsymbol{u}_j^{(\star,m)} = (u_j^{(1,m)}, u_j^{(2,m)}, \ldots, u_j^{(k,m)})$. Then for all $i \in \{1, 2, \ldots, k\}$, P_j sets his share of the Π_i-sharing, $\boldsymbol{X}_i$, to be $\boldsymbol{u}_j^{(i)} = (u_j^{(i,1)}, u_j^{(i,2)}, \ldots, u_j^{(i,\tilde{\ell})})$. All parties take $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_k$ as output.

of $\mathbf{0}$. Therefore, in the real world, $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ is a random Σ'-sharing of $\boldsymbol{u}_j^{(\star,m)}$ given the secret and the shares of corrupted parties. Since $\boldsymbol{u}_j^{(\star,m)}$ is determined by $\{\boldsymbol{\tau}_i, \boldsymbol{\rho}_i\}_{i=1}^k$, which are independent of the shares of $\{[\boldsymbol{r}_v]\}_{v=1}^{n\cdot\tilde{\ell}}$ and $\langle \boldsymbol{o}_j^{(m)} \rangle$ held by corrupted parties, $\boldsymbol{u}_j^{(\star,m)}$ is independent of the shares of $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ held by corrupted parties.

In the ideal world, recall that $\mathcal{S}$ learns the shares of $\{[\boldsymbol{r}_v]\}_{v=1}^{n\cdot\tilde{\ell}}$ held by corrupted parties in Step 2. And $\mathcal{S}$ learns the shares of $\langle \boldsymbol{o}_j^{(m)} \rangle$ held by corrupted parties in Step 3. Therefore, $\mathcal{S}$ can compute the shares of $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ held by corrupted parties. Also recall that we have shown that the secret, $\boldsymbol{u}_j^{(\star,m)}$, which is a part of $\{\boldsymbol{u}_j^{(i)}\}_{j\in\mathcal{C}orr, i\in\{1,\dots,k\}}$, has the same distribution as that in the real world. Therefore, the secret $\boldsymbol{u}_j^{(\star,m)}$ and the shares of $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ held by corrupted parties have the same distribution in both the ideal world and the real world. Finally, note that $\mathcal{S}$ samples a random Σ'-sharing $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ based on the secret $\boldsymbol{u}_j^{(\star,m)}$ simulated by $\mathcal{S}$ and the shares of $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ held by corrupted parties computed by $\mathcal{S}$. Thus, the shares of honest parties simulated by $\mathcal{S}$ have the same distribution as those in the real world.

Recall that all parties only communicate with other parties in Step 5. Therefore, the joint view of corrupted parties in the ideal world is identical to that in the real world.

- Finally, we show that the output of honest parties given the joint view of corrupted parties in the ideal world has the same distribution as that in the real world.

  Note that the joint view of corrupted parties is determined by (1) the shares of $[\boldsymbol{r}_v]$ held by corrupted parties for all $v \in \{1, 2, \dots, n \cdot \tilde{\ell}\}$, (2) the shares of $\langle \boldsymbol{o}_j^{(m)} \rangle$ held by corrupted parties for all $j \in \{1, 2, \dots, n\}$ and $m \in \{1, 2, \dots, \tilde{\ell}\}$, and (3) the shares of $\langle \boldsymbol{u}_j^{(\star,m)} \rangle$ held by honest parties for all $j \in \mathcal{C}orr$ and $m \in \{1, 2, \dots, \tilde{\ell}\}$. In the real world, (1) and (2) are independent of $\{\boldsymbol{\tau}_i, \boldsymbol{\rho}_i\}_{i=1}^k$ since Σ has threshold t. And (3) only depends on the shares $\{\boldsymbol{u}_j^{(i)}\}_{j\in\mathcal{C}orr, i\in\{1,\dots,k\}}$. Thus, in the real world, for all $i \in \{1, 2, \dots, k\}$, $\boldsymbol{X}_i = \mathtt{share}_i(\boldsymbol{\tau}_i, \boldsymbol{\rho}_i)$ is a random Π_i-sharing given the shares of corrupted parties. And honest parties simply output their shares of $\boldsymbol{X}_i$.

  In the ideal world, $\mathcal{S}$ perfectly simulates the shares of $\boldsymbol{X}_i = \mathtt{share}_i(\boldsymbol{\tau}_i, \boldsymbol{\rho}_i)$ held by corrupted parties and sends them to $\mathcal{F}_{\text{rand-sharing}}(\Pi_i)$. Then $\mathcal{F}_{\text{rand-sharing}}(\Pi_i)$ samples a random Π_i-sharing based on the shares of corrupted parties. The output of honest parties is their shares of $\boldsymbol{X}_i$ for all $i \in \{1, 2, \dots, k\}$.

  Therefore, the output of honest parties given the joint view of corrupted parties in the ideal world has the same distribution as that in the real world.

We conclude that Protocol RAND-SHARING securely computes $\{\mathcal{F}_{\text{rand-sharing}}(\Pi_i)\}_{i=1}^k$ in the $\{\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{randZero}}\}$-hybrid model against a semi-honest adversary who controls t parties. □

Cost of Protocol 11.3. We measure the amount of the preprocessing data and the ring elements that all parties need to send.

For the amount of preprocessing data, all parties need to prepare $n \cdot \tilde{\ell}$ random Σ-sharings. Let ℓ be the share size of Σ, i.e., the share space of Σ is $U = \mathcal{R}^\ell$. Thus, all Σ-sharings are of size $n^2 \cdot \tilde{\ell} \cdot \ell$ ring elements. All parties also need to prepare $n \cdot \tilde{\ell}$ random Σ'-sharings of $\mathbf{0} \in \mathcal{R}^k$. Let

ℓ' be the share size of Σ', i.e., the share space of Σ' is $U' = \mathcal{R}^{\ell'}$. Thus all Σ'-sharings are of size $n^2 \cdot \tilde{\ell} \cdot \ell'$ ring elements. In total, the amount of preprocessing data is $n^2 \cdot \tilde{\ell} \cdot (\ell + \ell')$ ring elements.

For the communication complexity, all parties only need to communicate in Step 5 of Protocol 11.3. The communication complexity is $n^2 \cdot \tilde{\ell} \cdot \ell'$ ring elements.

11.4 Instantiating Protocol RAND-SHARING via Packed Shamir Secret Sharing Scheme

For Large Finite Fields $\mathbb{F}$. Recall that when $k = (n-t+1)/2$, a degree-$(n-k)$ packed Shamir secret sharing has threshold t and is multiplication-friendly. Therefore, we use a degree-$(n-k)$ packed Shamir secret sharing scheme to instantiate Σ in Protocol RAND-SHARING. Then Σ' is a degree-$(n-1)$ packed Shamir secret sharing scheme. For Σ and Σ',

- The secret space is $\mathbb{F}^k$, where $k = (n-t+1)/2$.
- The share space is $\mathbb{F}$, i.e., each share is a single field element. Therefore $\ell = \ell' = 1$.

Thus, we obtain a protocol that prepares random sharings for $\Pi_1, \Pi_2, \ldots, \Pi_k$ with $2 \cdot n^2 \cdot \tilde{\ell} = O(n^2 \cdot \tilde{\ell})$ field elements of preprocessing data and $n^2 \cdot \tilde{\ell}$ field elements of communication. On average, the cost per random sharing is $O(\frac{n^2}{n-t+1} \cdot \tilde{\ell})$ field elements of both preprocessing data and communication. Note that when $t = (1-\epsilon) \cdot n$ for a positive constant ϵ, the achieved amortized cost per sharing is $O(n \cdot \tilde{\ell})$ field elements. In particular, $n \cdot \tilde{\ell}$ is the sharing size of Π_i for all $i \in \{1, 2, \ldots, k\}$. Essentially, it costs the same as letting a trusted party generate a random Π_i-sharing and distribute to all parties.

For Small Fields $\mathbb{F}_q$ and Rings $\mathbb{Z}/p^\ell\mathbb{Z}$. For a small field $\mathbb{F}_q$, we can use a large extension field of $\mathbb{F}_q$ so that the packed Shamir secret sharing scheme is available. For a ring $\mathbb{Z}/p^\ell\mathbb{Z}$, we can similarly use a large Galois ring of $\mathbb{Z}/p^\ell\mathbb{Z}$ so that the packed Shamir secret sharing scheme is available [1].

However, the approach of using a large extension field (or a large Galois ring) leads to a loss in the extension factor: While the share size grows up by the extension factor, the number of secrets that can be packed is still k. To resolve this issue, we can use the notion of reverse multiplication-friendly embedding (RMFE), which is first introduced in [21] for finite fields, and then extended to rings $\mathbb{Z}/p^\ell\mathbb{Z}$ in [27].

We take an RMFE over a finite field $\mathbb{F}_q$ as example, where q is an exponential of a prime. Informally, a pair of $\mathbb{F}_q$-linear maps (ϕ, ψ) is a $(r, m)_q$-reverse multiplication-friendly embedding if $\phi : \mathbb{F}_q^r \to \mathbb{F}_{q^m}$ and $\psi : \mathbb{F}_{q^m} \to \mathbb{F}_q^r$ satisfy that for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^r$, $\boldsymbol{x} * \boldsymbol{y} = \psi(\phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{y}))$. Intuitively, instead of doing a coordinate-wise multiplication over $\mathbb{F}_q$, an RMFE allows us to first map each vector (by ϕ) to an element in the extension field $\mathbb{F}_{q^m}$ and then perform a field multiplication over $\mathbb{F}_{q^m}$. The result can finally be transformed (by ψ) to the coordinate-wise multiplication between the input two vectors.

Now consider an arithmetic secret sharing scheme Σ over $\mathbb{F}_q$:

- The secret space is $Z = \mathbb{F}_q^{k \cdot r}$.
- The share space is $U = \mathbb{F}_{q^m}$, which can be viewed as $\mathbb{F}_q^m$.

- For $\boldsymbol{x} \in Z$, the sharing of $\boldsymbol{x}$ is computed as follows: We first divide $\boldsymbol{x}$ into k vectors of dimension r. Then we map each vector to a field element in $\mathbb{F}_{q^m}$ by using ϕ. Next we use a degree-$(n-k)$ packed Shamir secret sharing scheme over $\mathbb{F}_{q^m}$ to store the k elements in $\mathbb{F}_{q^m}$.
- For a sharing $\boldsymbol{X}$, we first view it as a degree-$(n-k)$ packed Shamir sharing over $\mathbb{F}_{q^m}$ and reconstruct its secret $\boldsymbol{s} \in \mathbb{F}_{q^m}^k$. To recover the secret, we apply ϕ^{-1} on each element in $\boldsymbol{s}$. (See [62] for an explicit construction of ϕ^{-1}.)

Note that Σ have threshold t due to the use of the degree-$(n-k)$ packed Shamir secret sharing scheme. Also note that Σ is still multiplication-friendly: To multiply a constant vector $\boldsymbol{c} \in \mathbb{F}_q^{k \cdot r}$ with a Σ-sharing of $\boldsymbol{x}$, we first transform $\boldsymbol{c}$ to a degree-$(k-1)$ packed Shamir sharing over $\mathbb{F}_{q^m}$ similarly as above. Then after multiplying the two packed Shamir sharings, we can reconstruct the secret $\boldsymbol{c} * \boldsymbol{x}$ by using ψ to decode the secrets of the resulting packed Shamir sharing.

Therefore, relying on packed Shamir secret sharing schemes and RMFEs, we can use Σ to instantiate Protocol RAND-SHARING for a small field $\mathbb{F}_q$. For both Σ and Σ',

- The secret space is $\mathbb{F}_q^{k \cdot r}$, where $k = (n - t + 1)/2$.
- The share space is $\mathbb{F}_q^m$, i.e., each share is m field elements. Therefore $\ell = \ell' = m$.

Note that, while the share size of Σ and Σ' grows up by a factor of m, the number of secrets that can be packed becomes $k \cdot r$, which grows up by a factor of r. Thus, we obtain a protocol that prepares random sharings for arbitrary $\mathbb{F}_q$-arithmetic secret sharing schemes $\Pi_1, \Pi_2, \ldots, \Pi_{k \cdot r}$ with $2 \cdot n^2 \cdot \tilde{\ell} \cdot m = O(n^2 \cdot \tilde{\ell} \cdot m)$ field elements of preprocessing data, and $n^2 \cdot \tilde{\ell} \cdot m$ field elements of communication. On average, the cost per random sharing is $O(\frac{n^2}{n-t+1} \cdot \tilde{\ell} \cdot \frac{m}{r})$ field elements of both preprocessing data and communication.

In [21], Cascudo, et al show that for all $\mathbb{F}_q$, there exists a family of RMFEs with r slowly grows to infinity and m/r is bounded by a constant. Thus, the amortized cost per random sharing becomes $O(\frac{n^2}{n-t+1} \cdot \tilde{\ell})$ field elements, which is the same as that for a large finite field.

A similar result for rings $\mathbb{Z}/p^\ell\mathbb{Z}$ can be achieved by using RMFEs over rings $\mathbb{Z}/p^\ell\mathbb{Z}$ constructed in [27].

11.5 Application of $\mathcal{F}_{\text{rand-sharing}}$

Let Σ and Σ' be two threshold-t $\mathcal{R}$-arithmetic secret sharing schemes. Let $f : Z \to Z'$ be an $\mathcal{R}$-module homomorphism, where Z and Z' are the secret spaces of Σ and Σ' respectively. Suppose given a Σ-sharing, $\boldsymbol{X}$, all parties want to compute a Σ'-sharing, $\boldsymbol{Y}$, subject to $\mathtt{rec}'(\boldsymbol{Y}) = f(\mathtt{rec}(\boldsymbol{X}))$, where $\mathtt{rec}$ and $\mathtt{rec}'$ are reconstruction maps of Σ and Σ', respectively. We refer to this problem as sharing transformation.

As discussed in Chapter 10, sharing transformation can be efficiently solved with the help of a pair of random sharings $(\boldsymbol{R}, \boldsymbol{R}')$, where $\boldsymbol{R}$ is a Σ-sharing, and $\boldsymbol{R}'$ is a Σ'-sharing subject to $\mathtt{rec}'(\boldsymbol{R}') = f(\mathtt{rec}(\boldsymbol{R}))$. Consider the following $\mathcal{R}$-arithmetic secret sharing scheme $\widetilde{\Sigma} = \widetilde{\Sigma}(\Sigma, \Sigma', f)$:

- The secret space is Z, the same as that of Σ.
- The share space is $U \times U'$, where U is the share space of Σ and U' is the share space of Σ'.

- For a secret $\boldsymbol{x} \in Z$, the sharing of $\boldsymbol{x}$ is the concatenation of a Σ-sharing of $\boldsymbol{x}$ and a Σ'-sharing of $f(\boldsymbol{x})$.
- For a sharing $\boldsymbol{X}$, recall that each share of $\widetilde{\Sigma}$ consists of one share of Σ and one share of Σ'. The secret of $\boldsymbol{X}$ can be recovered by applying $\texttt{rec}$ of Σ on the sharing which consists of the shares of Σ in $\boldsymbol{X}$.

Then, $(\boldsymbol{R}, \boldsymbol{R}')$ is a random $\widetilde{\Sigma}$-sharing. The problem is reduced to prepare a random $\widetilde{\Sigma}$-sharing, which can be done by $\mathcal{F}_{\text{rand-sharing}}(\widetilde{\Sigma})$.

We summarize the functionality $\mathcal{F}_{\text{tran}}$ in Functionality 11.4 and the protocol TRAN for $\mathcal{F}_{\text{tran}}$ in Protocol 11.5.

Figure 11.4: Functionality $\mathcal{F}_{\text{tran}}$

1. $\mathcal{F}_{\text{tran}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$. $\mathcal{F}_{\text{tran}}$ also receives two threshold-t $\mathcal{R}$-arithmetic secret sharing schemes Σ, Σ' and an $\mathcal{R}$-module homomorphism $f : Z \to Z'$.
2. $\mathcal{F}_{\text{tran}}$ receives a Σ-sharing $\boldsymbol{X}$ from all parties and computes $f(\texttt{rec}(\boldsymbol{X}))$.
3. $\mathcal{F}_{\text{tran}}$ receives from the adversary a set of shares $\{\boldsymbol{u}'_j\}_{j\in\mathcal{C}orr}$, where $\boldsymbol{u}'_j \in U'$ for all $P_j \in \mathcal{C}orr$.
4. $\mathcal{F}_{\text{tran}}$ samples a random Σ'-sharing, $\boldsymbol{Y}$, such that $\texttt{rec}'(\boldsymbol{Y}) = f(\texttt{rec}(\boldsymbol{X}))$ and the shares of corrupted parties are identical to those received from the adversary, i.e., $\pi_{\mathcal{C}orr}(\boldsymbol{Y}) = (\boldsymbol{u}'_j)_{j\in\mathcal{C}orr}$. If such a sharing does not exist, $\mathcal{F}_{\text{tran}}$ sends $\texttt{abort}$ to honest parties and halts.
5. Otherwise, $\mathcal{F}_{\text{tran}}$ distributes the shares of $\boldsymbol{Y}$ to honest parties.

Figure 11.5: Protocol TRAN

1. Let Σ, Σ' be two threhsold-t $\mathcal{R}$-arithmetic secret sharing schemes and $f : Z \to Z'$ be an $\mathcal{R}$-module homomorphism. All parties hold a Σ-sharing, $\boldsymbol{X}$, at the beginning of the protocol.
2. Let $\widetilde{\Sigma} = \widetilde{\Sigma}(\Sigma, \Sigma', f)$ be the threshold-t $\mathcal{R}$-arithmetic secret sharing scheme defined above. All parties invoke $\mathcal{F}_{\text{rand-sharing}}(\widetilde{\Sigma})$ and obtain a $\widetilde{\Sigma}$-sharing $(\boldsymbol{R}, \boldsymbol{R}')$.
3. All parties locally compute $\boldsymbol{X} + \boldsymbol{R}$ and send their shares to the first party P_1.
4. P_1 reconstructs the secret of $\boldsymbol{X} + \boldsymbol{R}$, denoted by $\boldsymbol{w}$. Then P_1 computes $f(\boldsymbol{w})$ and generates a Σ'-sharing of $f(\boldsymbol{w})$, denoted by $\boldsymbol{W}$. Finally, P_1 distributes the shares of $\boldsymbol{W}$ to all parties.
5. All parties locally compute $\boldsymbol{Y} = \boldsymbol{W} - \boldsymbol{R}'$.

Lemma 11.3. *For all threshold-t $\mathcal{R}$-arithmetic secret sharing schemes Σ, Σ' and for all $\mathcal{R}$-module homomorphism $f : Z \to Z'$, Protocol* TRAN *securely computes $\mathcal{F}_{tran}$ in the $\mathcal{F}_{rand\text{-}sharing}$-hybrid model against a semi-honest adversary who controls t parties.*

We obtain the following theorem for our sharing transformation protocol.

Theorem 11.1. *Let n be the number of parties, t be the number of corrupted parties, and $k = (n-t+1)/2$. Let $\mathbb{F}$ be a finite field of size $2n$. Let ℓ_1, ℓ_2 be two positive integers. For all k tuples $\{(\Sigma_i, \Sigma'_i, f_i)\}_{i=1}^k$ and for all $\{\boldsymbol{X}_i\}_{i=1}^k$ such that Σ_i, Σ'_i are $\mathbb{F}$-linear secret sharing schemes with injective sharing functions and with share size ℓ_1, ℓ_2, f_i is a linear map from the secret space of Σ_i to that of Σ'_i, and $\boldsymbol{X}_i$ is a Σ_i-sharing held by all parties, there is an information-theoretic MPC protocol with semi-honest security against t corrupted parties that transforms $\boldsymbol{X}_i$ to a Σ'-sharing $\boldsymbol{Y}_i$ such that the secret of $\boldsymbol{Y}_i$ is equal to the result of applying f_i on the secret of $\boldsymbol{X}_i$. The cost of the protocol is $O(n^2 \cdot (\ell_1 + \ell_2))$ elements of preprocessing data and $O(n^2 \cdot (\ell_1 + \ell_2))$ elements of communication.*

Remark 11.1. *We note that the preprocessing phase only prepares random degree-$(n-k)$ packed Shamir sharings and random degree-$(n-1)$ packed Shamir sharings of $\mathbf{0} \in \mathbb{F}^k$. With instantiation in Chapter 15.1, the preprocessing can be done with communication complexity $O(\frac{n^3}{k} \cdot (\ell_1 + \ell_2))$ field elements for k sharing transformations.*

Improvement of Protocol RAND-SHARING for a Concrete Sharing Transformation. Our MPC protocol needs to perform the following sharing transformation. Let $\mathbb{F}$ be a large finite field. Recall that for a packed Shamir secret sharing scheme over $\mathbb{F}$, we use fixed $\alpha_1, \ldots, \alpha_n$ for shares of all parties. For two (potentially different) vectors of field elements $\mathtt{pos} = (p_1, \ldots, p_k)$ and $\mathtt{pos}' = (p'_1, \ldots, p'_k)$ such that they are disjoint with $\{\alpha_1, \ldots, \alpha_n\}$, all parties start with a degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{x} \| \mathtt{pos}]_{n-1}$. They want to compute a degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{y} \| \mathtt{pos}']_{n-k}$ such that for all $i \in \{1, 2, \ldots, k\}$, y_i is equal to x_j for some j. In particular, it is possible that for two different indices i and i', y_i and $y_{i'}$ are equal to the same x_j.

Let $f : \mathbb{F}^k \to \mathbb{F}^k$ be an $\mathbb{F}$-linear map which satisfies that $\boldsymbol{y} = f(\boldsymbol{x})$. Then, we need to prepare a pair of random sharings $([\boldsymbol{r} \| \mathtt{pos}]_{n-1}, [f(\boldsymbol{r}) \| \mathtt{pos}']_{n-k})$. In particular, we can view it as a random sharing of the following $\mathbb{F}$-arithmetic secret sharing scheme Π:

- The secret space $Z = \mathbb{F}^k$.
- The share space $U = \mathbb{F}^2$.
- For a secret $\boldsymbol{x} \in Z$, the sharing of $\boldsymbol{x}$ is $([\boldsymbol{x} \| \mathtt{pos}]_{n-1}, [f(\boldsymbol{x}) \| \mathtt{pos}']_{n-k})$.
- For a sharing $\boldsymbol{X} = (\boldsymbol{X}_1, \boldsymbol{X}_2)$, we view $\boldsymbol{X}_1$ as a degree-$(n-1)$ packed Shamir secret sharing scheme and reconstruct the secret $\boldsymbol{x}$.

We note that when directly using RAND-SHARING (instantiated by packed Shamir secret sharing schemes, see Chapter 11.4) to prepare a random Π-sharing, the communication complexity per sharing is $\frac{2n^2}{k}$ elements. This is because in Step 5 of RAND-SHARING, we need to compute $\tilde{\ell} = 2$ Σ'-sharings for the shares of each party, where $\tilde{\ell}$ is the share size of Π. These sharings are then reconstructed to their corresponding holders, which incurs $2n^2$ elements of communication for k sharings.

We observe that a random degree-d packed Shamir secret sharing scheme has the following

nice properties:

- The shares of the first $d+1$ parties are uniformly random.
- The secret and the shares of the rest of $n-d-1$ parties are determined by the shares of the first $d+1$ parties.

When the secret of a random degree-d packed Shamir sharing is given,

- The shares of the first $d-k+1$ parties are uniformly random.
- The shares of the rest of $n-d+k-1$ parties are determined by the secret and the shares of the first $d+1$ parties.

Thus, when preparing a random degree-d packed Shamir sharing, we can view the shares of the first $d+1$ parties as the random tape for generating the whole sharing. In particular, the secret will be determined by the shares of the first $d+1$ parties. In Protocol RAND-SHARING, all parties invoke $\mathcal{F}_{\text{rand}}$ to obtain random Σ-sharings for the random tapes. The secrets of these Σ-sharings can directly be viewed as the shares of the first $d+1$ parties. Thus, we can let the first $d+1$ parties reconstruct their shares in the preprocessing phase. In this way, we only need to reconstruct shares for the rest of $n-d-1$ parties in the online phase. Similarly, when preparing a random degree-d packed Shamir sharing for a given input $\boldsymbol{x}$, we can view the shares of the first $d-k+1$ parties as the random tape for generating the whole sharing. We can let the first $d-k+1$ parties reconstruct their shares in the preprocessing phase. In this way we only need to reconstruct shares for the rest of $n-d+k-1$ parties in the online phase.

For the sharing transformation we are interested in, we need to prepare random sharings in the form of $([\boldsymbol{r}\|\mathtt{pos}]_{n-1}, [f(\boldsymbol{r})\|\mathtt{pos}']_{n-k})$. For the first sharing, it is a random degree-$(n-1)$ packed Shamir sharing. We can let all parties reconstruct their shares in the preprocessing phase. For the second sharing, it is a random degree-$(n-k)$ packed Shamir sharing given the secret $f(\boldsymbol{r})$. We can let $n-2k+1$ parties reconstruct their shares in the preprocessing phase. Thus, in the online phase, we only need to reconstruct the shares of $[f(\boldsymbol{r})\|\mathtt{pos}']_{n-k}$ of the last $2k-1$ parties.

Concretely, in the preprocessing phase, all parties prepare n random Σ-sharings. The i-th sharing is reconstructed to P_i, and P_i takes the secret as its shares of the degree-$(n-1)$ packed Shamir sharings in $\Pi_1, \ldots, \Pi_k$. (Recall that each time we prepare random sharings for k arithmetic secret sharing schemes. Here we assume that $\Pi_1, \ldots, \Pi_k$ are all in the form of Π constructed above.) Then, all parties prepare $n-2k+1$ random Σ-sharings. The i-th sharing is reconstructed to P_i, and P_i takes the secret as its shares of the degree-$(n-k)$ packed Shamir sharings in $\Pi_1, \ldots, \Pi_k$. After that, all parties prepare $2k-1$ Σ'-sharing of $\mathbf{0} \in \mathbb{F}^k$. Thus, the preprocessing data consists of $2n-2k+1$ random Σ sharings and $2k-1$ random Σ'-sharings of $\mathbf{0} \in \mathbb{F}^k$. Since each Σ-sharing is also reconstructed to a single party, the amount of preprocessing data is $2(2n-2k+1) \cdot n + (2k-1) \cdot n < 4n^2$ field elements.

In the online phase, all parties use the $2n-2k+1$ random Σ-sharings to compute Σ'-sharings of the shares of the last $2k-1$ parties of the degree-$(n-k)$ packed Shamir sharings in $\Pi_1, \ldots, \Pi_k$. Then they use random Σ'-sharings of $\mathbf{0}$ to mask these sharings and let the last $2k-1$ parties reconstruct their shares. Thus, the communication complexity is $(2k-1) \cdot n < 2k \cdot n$ field elements.

Thus, with this improvement, we obtain a protocol that prepares a random sharing for Π with $4n^2/k$ field elements of preprocessing data, and $2n$ field elements of communication. When we

use TRAN to perform the above sharing transformation, the amortized cost of TRAN is $4n^2/k$ field elements of preprocessing data and $4n$ elements of communication.

Chapter 12

Semi-Honest Protocol

In this chapter, we focus on the semi-honest security. We show how to use packed Shamir sharing schemes and $\mathcal{F}_{\text{tran}}$ (introduced in Chapter 11.5) to evaluate a circuit against a semi-honest adversary who controls t parties. We focus on a finite field $\mathbb{F}$ of size $|\mathbb{F}| \geq |C| + n$, where $|C|$ is the circuit size. Let $k = (n - t + 1)/2$.

Recall that we use $[\boldsymbol{x}\|\texttt{pos}]_d$ to represent a degree-d packed Shamir sharing of $\boldsymbol{x} \in \mathbb{F}^k$ stored at positions $\texttt{pos} = (p_1, p_2, \ldots, p_k)$. Also recall that the shares of a degree-d packed Shamir sharing are at evaluation points $\alpha_1, \alpha_2, \ldots, \alpha_n$. Let $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_k)$ be k distinct elements in $\mathbb{F}$ that are different from $(\alpha_1, \alpha_2, \ldots, \alpha_n)$. We use $\boldsymbol{\beta}$ as the default positions for a degree-d packed Shamir sharing, and simply write $[\boldsymbol{x}]_d = [\boldsymbol{x}\|\boldsymbol{\beta}]_d$.

12.1 Circuit-Independent Preprocessing Phase

In the circuit-independent preprocessing phase, all parties need to prepare packed Beaver triples. For every group of k multiplication gates, all parties prepare a packed Beaver triple

$$([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-k})$$

where $\boldsymbol{a}, \boldsymbol{b}$ are random vectors in $\mathbb{F}^k$ and $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. We will use the technique of packed Beaver triples to compute multiplication gates in the online phase. The functionality $\mathcal{F}_{\text{prep}}$ for the circuit independent preprocessing phase appears in Functionality 12.1.

12.2 Online Computation Phase

Recall that for the field size it holds that $|\mathbb{F}| \geq |C| + n$, where $|C|$ is the circuit size. Let $\beta_1, \beta_2, \ldots, \beta_{|C|}$ be $|C|$ distinct field elements that are different from $\alpha_1, \alpha_2, \ldots, \alpha_n$. (Recall that we have already defined $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_k)$, which are used as the default positions for a packed Shamir sharing.) We associate the field element β_i with the i-th gate in C. We will use β_i as the position to store the output value of the i-th gate in a degree-$(n-k)$ packed Shamir sharing.

Concretely, for each layer, gates that have the same type are divided into groups of size k. For each group of k gates, all parties will compute a degree-$(n-k)$ packed Shamir sharing such that the results are stored at the positions associated with these k gates respectively.

Figure 12.1: Functionality $\mathcal{F}_{\text{prep}}$

For every group of k multiplication gates:

1. $\mathcal{F}_{\text{prep}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$.
2. $\mathcal{F}_{\text{prep}}$ receives from the adversary a set of shares $\{(a_j, b_j, c_j)\}_{j\in\mathcal{C}orr}$. $\mathcal{F}_{\text{prep}}$ samples two random vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^k$ and computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Then $\mathcal{F}_{\text{prep}}$ computes three degree-$(n-k)$ packed Shamir sharings $[\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-k}$ such that for all $P_j \in \mathcal{C}orr$, the j-th share of $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-k})$ is (a_j, b_j, c_j).
3. $\mathcal{F}_{\text{prep}}$ distributes the shares of $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-k})$ to honest parties.

12.2.1 Input Layer

In the input layer, input gates are divided into groups of size k based on the input holders. For a group of k input gates belonging to the same client, suppose $\boldsymbol{x}$ are the inputs, and $\texttt{pos} = (p_1, p_2, \ldots, p_k)$ are the positions associated with these k gates. The client generates a random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{x}\|\texttt{pos}]_{n-k}$ and distributes the shares to all parties.

12.2.2 Network Routing

In each intermediate layer, all gates are divided into groups of size k based on their types (i.e., multiplication gates or addition gates). For a group of k gates, all parties prepare two degree-$(n-k)$ packed Shamir sharings, one for the first inputs of all gates, and the other one for the second inputs of all gates.

Concretely, for a group k gates in the current layer, suppose $\boldsymbol{x}$ are the first inputs of these k gates, and $\boldsymbol{y}$ are the second inputs of these k gates. All parties will prepare two degree-$(n-k)$ packed Shamir sharings $[\boldsymbol{x}]_{n-k}$ and $[\boldsymbol{y}]_{n-k}$ stored at the default positions. The reason of choosing the default positions is to use the packed Beaver triples all parties have prepared in the preprocessing phase. Recall that the packed Beaver triples all use the default positions. In the following, we focus on inputs $\boldsymbol{x}$.

Collecting Secrets from Previous Layers. Let $x'_1, x'_2, \ldots, x'_{\ell_1}$ be the different values in $\boldsymbol{x}$ from previous layers. Let $c_1, c_2, \ldots, c_{\ell_2}$ be the constant values in $\boldsymbol{x}$. Then $\ell_1 + \ell_2 \leq k$. For each of the rest of $k - \ell_1 - \ell_2$ values in $\boldsymbol{x}$, it is the same as x'_i for some $i \in \{1, 2, \ldots, \ell_1\}$. In this step, we will prepare a degree-$(n-1)$ packed Shamir sharing that contains the secrets $x'_1, x'_2, \ldots, x'_{\ell_1}$ and $c_1, c_2, \ldots, c_{\ell_2}$.

Note that $\{x'_i\}_{i=1}^{\ell_1}$ are the output values of ℓ_1 different gates in previous layers. Let $p_1, p_2, \ldots, p_{\ell_1}$ be the positions associated with these ℓ_1 gates. We choose another arbitrary $k-\ell_1$ different positions $p_{\ell_1+1}, \ldots, p_k$ which are also different from $\alpha_1, \alpha_2, \ldots, \alpha_n$, and set $\texttt{pos} = (p_1, p_2, \ldots, p_k)$. Suppose for all $1 \leq i \leq \ell_1$, $[\boldsymbol{x}^{(i)}\|\texttt{pos}^{(i)}]_{n-k}$ is the degree-$(n-k)$ packed Shamir sharing from some previous layer that contains the secret x'_i stored at position p_i.

Let $\boldsymbol{e}_i$ be the i-th unit vector in $\mathbb{F}^k$ (i.e., only the i-th term is 1 and all other terms are 0). All parties locally compute a degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{e}_i\|\texttt{pos}]_{k-1}$. Let $\boldsymbol{x}' =$

$(x'_1, \ldots, x'_{\ell_1}, c_1, \ldots, c_{\ell_2}, 0, \ldots, 0)$ be a vector in $\mathbb{F}^k$. Then all parties locally compute

$$\sum_{i=1}^{\ell_1} [\boldsymbol{e}_i \| \mathtt{pos}]_{k-1} \cdot [\boldsymbol{x}^{(i)} \| \mathtt{pos}^{(i)}]_{n-k} + \sum_{i=1}^{\ell_2} c_i \cdot [\boldsymbol{e}_{\ell_1+i} \| \mathtt{pos}]_{k-1}.$$

We show that this is a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{x}'$ stored at positions $\mathtt{pos}$. It is clear that the resulting sharing has degree $n-1$. We only need to show the following three points:

- For all $1 \leq j \leq \ell_1$, the secret stored at position p_j is equal to x'_j.
- For all $\ell_1 + 1 \leq j \leq \ell_1 + \ell_2$, the secret stored at position p_j is equal to $c_{j-\ell_1}$.
- For all $\ell_1 + \ell_2 + 1 \leq j \leq k$, the secret stored at position p_j is equal to 0.

For all $1 \leq i \leq \ell_1 + \ell_2$, let f_i be the polynomial corresponding to $[\boldsymbol{e}_i \| \mathtt{pos}]_{k-1}$. For all $1 \leq i \leq \ell_1$, let g_i be the polynomial corresponding to $[\boldsymbol{x}^{(i)} \| \mathtt{pos}^{(i)}]_{n-k}$. Then the polynomial corresponding to the resulting sharing is $h = \sum_{i=1}^{\ell_1} f_i \cdot g_i + \sum_{i=1}^{\ell_2} c_i \cdot f_{\ell_1+i}$.

Note that f_i satisfies that $f_i(p_i) = 1$ and $f_i(p_j) = 0$ for all $j \neq i$. And g_i satisfies that $g_i(p_i) = x'_i$. Therefore, for all $1 \leq j \leq \ell_1$,

$$h(p_j) = \sum_{i=1}^{\ell_1} f_i(p_j) \cdot g_i(p_j) + \sum_{i=1}^{\ell_2} c_i \cdot f_{\ell_1+i}(p_j) = f_j(p_j) \cdot g_j(p_j) = x'_j.$$

For all $\ell_1 + 1 \leq j \leq \ell_2$,

$$h(p_j) = \sum_{i=1}^{\ell_1} f_i(p_j) \cdot g_i(p_j) + \sum_{i=1}^{\ell_2} c_i \cdot f_{\ell_1+i}(p_j) = c_{j-\ell_1} \cdot f_j(p_j) = c_{j-\ell_1}.$$

For all $\ell_1 + \ell_2 + 1 \leq j \leq k$,

$$h(p_j) = \sum_{i=1}^{\ell_1} f_i(p_j) \cdot g_i(p_j) + \sum_{i=1}^{\ell_2} c_i \cdot f_{\ell_1+i}(p_j) = 0.$$

Thus, the resulting sharing is a degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{x}'$ stored at positions $\mathtt{pos}$, denoted by $[\boldsymbol{x}' \| \mathtt{pos}]_{n-1}$.

Transforming to the Desired Sharing. Now all parties hold a degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{x}' \| \mathtt{pos}]_{n-1}$. Recall that $\boldsymbol{x}'$ contains all different values in $\boldsymbol{x}$ from previous layers and all constant values. For each of the rest of values in $\boldsymbol{x}$, it is the same as x'_i for some $i \in \{1, 2, \ldots, \ell_1\}$. Then there is a linear map $f : \mathbb{F}^k \to \mathbb{F}^k$ such that $\boldsymbol{x} = f(\boldsymbol{x}')$. Recall that $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_k)$ are the default positions. Let Σ be the degree-$(n-1)$ packed Shamir secret sharing scheme that stores secrets at positions $\mathtt{pos}$. Let Σ' be the degree-$(n-k)$ packed Shamir secret sharing scheme that stores secrets at positions $\boldsymbol{\beta}$. Then $[\boldsymbol{x}' \| \mathtt{pos}]_{n-1}$ is a Σ-sharing, and the sharing we want to prepare, $[\boldsymbol{x}]_{n-k} = [\boldsymbol{x} \| \boldsymbol{\beta}]_{n-k}$, is a Σ'-sharing with $\boldsymbol{x} = f(\boldsymbol{x}')$.

All parties invoke $\mathcal{F}_{\text{tran}}$ with (Σ, Σ', f) and $[\boldsymbol{x}' \| \mathtt{pos}]_{n-1}$, and obtain $[\boldsymbol{x}]_{n-k}$.

Figure 12.2: Protocol NETWORK

1. Suppose all parties want to prepare a degree-$(n-k)$ packed Shamir sharing of $\boldsymbol{x}$ stored at the default positions $\boldsymbol{\beta}$.
2. Let $x'_1, x'_2, \ldots, x'_{\ell_1}$ be the different wire values in $\boldsymbol{x}$ from previous layers. Let $c_1, c_2, \ldots, c_{\ell_2}$ be the constant values in $\boldsymbol{x}$. Let $\boldsymbol{x}' = (x'_1, \ldots, x'_{\ell_1}, c_1, \ldots, c_{\ell_2}, 0, \ldots, 0) \in \mathbb{F}^k$.
3. For all $1 \leq i \leq \ell_1$, let $[\boldsymbol{x}^{(i)} \| \mathtt{pos}^{(i)}]_{n-k}$ be the degree-$(n-k)$ packed Shamir sharing from some previous layer that contains the secret x'_i stored at position p_i. Let $p_{\ell_1+1}, \ldots, p_k$ be the first $k - \ell_1$ distinct positions that are different from $p_1, \ldots, p_{\ell_1}$ and $\alpha_1, \ldots, \alpha_n$. Let $\mathtt{pos} = (p_1, \ldots, p_k)$.
4. Let $\boldsymbol{e}_i$ be the i-th unit vector in $\mathbb{F}^k$ (i.e., only the i-th term is 1 and all other terms are 0). All parties locally compute a degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{e}_i \| \mathtt{pos}]_{k-1}$.
5. All parties locally compute

$$[\boldsymbol{x}' \| \mathtt{pos}]_{n-1} = \sum_{i=1}^{\ell_1} [\boldsymbol{e}_i \| \mathtt{pos}]_{k-1} \cdot [\boldsymbol{x}^{(i)} \| \mathtt{pos}^{(i)}]_{n-k} + \sum_{i=1}^{\ell_2} c_i \cdot [\boldsymbol{e}_{\ell_1+i} \| \mathtt{pos}]_{k-1}.$$

6. Let $f : \mathbb{F}^k \to \mathbb{F}^k$ be a linear map such that $\boldsymbol{x} = f(\boldsymbol{x}')$. Let Σ be the degree-$(n-1)$ packed Shamir secret sharing scheme that stores secrets at positions $\mathtt{pos}$. Let Σ' be the degree-$(n-k)$ packed Shamir secret sharing scheme that stores secrets at positions $\boldsymbol{\beta}$. All parties invoke $\mathcal{F}_{\text{tran}}$ with (Σ, Σ', f) and $[\boldsymbol{x}' \| \mathtt{pos}]_{n-1}$, and output $[\boldsymbol{x}]_{n-k}$.

Summary of Network Routing. We describe the protocol NETWORK of preparing an input degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{x}]_{n-k}$ in Protocol 12.2.

12.2.3 Evaluating Addition Gates and Multiplication Gates

Addition Gates. For a group of k addition gates, recall that all parties have prepared two degree-$(n-k)$ packed Shamir sharings $[\boldsymbol{x}]_{n-k}, [\boldsymbol{y}]_{n-k}$ where $\boldsymbol{x}$ are the first inputs of these k gates, and $\boldsymbol{y}$ are the second inputs of these k gates. The description of PACKED-ADD appears in Protocol 12.3. Note that in Step 3 of Protocol PACKED-ADD, we use the fact that a degree-$(n-k)$ packed Shamir sharing can be viewed as a degree-$(n-1)$ packed Shamir sharing.

Multiplication Gates. For a group of k multiplication gates, recall that all parties have prepared two degree-$(n-k)$ packed Shamir sharings $[\boldsymbol{x}]_{n-k}, [\boldsymbol{y}]_{n-k}$ where $\boldsymbol{x}$ are the first inputs of these k gates, and $\boldsymbol{y}$ are the second inputs of these k gates. Let $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-k})$ be the packed Beaver triple prepared in the preprocessing phase. We will use the technique of packed Beaver triples to evaluate multiplication gates. The description of PACKED-MULT appears in Protocol 12.4.

Figure 12.3: Protocol PACKED-ADD

1. Suppose $[\boldsymbol{x}]_{n-k}, [\boldsymbol{y}]_{n-k}$ are the input packed Shamir sharings of the addition gates.
2. All parties locally compute $[\boldsymbol{z}]_{n-k} = [\boldsymbol{x}]_{n-k} + [\boldsymbol{y}]_{n-k}$.
3. Suppose $\mathtt{pos} = (p_1, p_2, \ldots, p_k)$ are the positions associated with these k addition gates. Recall that $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_k)$ are the default positions. Let Σ be the degree-$(n-1)$ packed Shamir secret sharing scheme that stores secrets at positions $\boldsymbol{\beta}$. Let Σ' be the degree-$(n-k)$ packed Shamir secret sharing scheme that stores secrets at positions $\mathtt{pos}$. Let $I : \mathbb{F}^k \to \mathbb{F}^k$ be the identity map.

   All parties invoke $\mathcal{F}_{\text{tran}}$ with (Σ, Σ', I) and $[\boldsymbol{z}]_{n-k}$, and output $[\boldsymbol{z} \| \mathtt{pos}]_{n-k}$.

Figure 12.4: Protocol PACKED-MULT

1. Suppose $[\boldsymbol{x}]_{n-k}, [\boldsymbol{y}]_{n-k}$ are the input packed Shamir sharings of the multiplication gates. All parties will use a fresh random packed Beaver triple $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-k})$ prepared in the preprocessing phase.
2. All parties locally compute $[\boldsymbol{x} + \boldsymbol{a}]_{n-k} = [\boldsymbol{x}]_{n-k} + [\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{y} + \boldsymbol{b}]_{n-k} = [\boldsymbol{y}]_{n-k} + [\boldsymbol{b}]_{n-k}$.
3. The first party P_1 collects the whole sharings $[\boldsymbol{x} + \boldsymbol{a}]_{n-k}, [\boldsymbol{y} + \boldsymbol{b}]_{n-k}$ and reconstructs the secrets $\boldsymbol{x} + \boldsymbol{a}, \boldsymbol{y} + \boldsymbol{b}$. Then, P_1 computes the sharings $[\boldsymbol{x} + \boldsymbol{a}]_{k-1}, [\boldsymbol{y} + \boldsymbol{b}]_{k-1}$ and distributes the shares to other parties.
4. All parties locally compute

   $$[\boldsymbol{z}]_{n-1} := [\boldsymbol{x} + \boldsymbol{a}]_{k-1} \cdot [\boldsymbol{y} + \boldsymbol{b}]_{k-1} - [\boldsymbol{x} + \boldsymbol{a}]_{k-1} \cdot [\boldsymbol{b}]_{n-k} - [\boldsymbol{y} + \boldsymbol{b}]_{k-1} \cdot [\boldsymbol{a}]_{n-k} + [\boldsymbol{c}]_{n-k}.$$

5. Suppose $\mathtt{pos} = (p_1, p_2, \ldots, p_k)$ are the positions associated with these k multiplication gates. Recall that $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_k)$ are the default positions. Let Σ be the degree-$(n-1)$ packed Shamir secret sharing scheme that stores secrets at positions $\boldsymbol{\beta}$. Let Σ' be the degree-$(n-k)$ packed Shamir secret sharing scheme that stores secrets at positions $\mathtt{pos}$. Let $I : \mathbb{F}^k \to \mathbb{F}^k$ be the identity map.

   All parties invoke $\mathcal{F}_{\text{tran}}$ with (Σ, Σ', I) and $[\boldsymbol{z}]_{n-1}$, and output $[\boldsymbol{z} \| \mathtt{pos}]_{n-k}$.

12.2.4 Output Layer

In the output layer, output gates are divided into groups of size k based on the output receivers. For a group of k output gates belonging to the same client, suppose $\boldsymbol{x}$ are the inputs. All parties invoke the protocol NETWORK to prepare $[\boldsymbol{x}]_{n-k}$. Then, all parties send their shares to the client to allow him to reconstruct the output.

12.2.5 Main Protocol

Given the above protocols the main semi-honest protocol follows in a straightforward way. We recall the Functionality $\mathcal{F}_{\text{main}}$ in Functionality 6.13 and describe the main protocol PACKED-MAIN in Protocol 12.5.

Figure 6.13: Functionality $\mathcal{F}_{\text{main}}$

1. $\mathcal{F}_{\text{main}}$ receives from all clients their inputs.
2. $\mathcal{F}_{\text{main}}$ evaluates the circuit and computes the output. $\mathcal{F}_{\text{main}}$ distributes the output to all clients.

Lemma 12.1. *Protocol* PACKED-MAIN *securely computes $\mathcal{F}_{main}$ in the $(\mathcal{F}_{prep}, \mathcal{F}_{tran})$-hybrid model against a semi-honest adversary who controls t parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}orr$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties.

The correctness of PACKED-MAIN follows from (1) the correctness of the protocol NETWORK for network routing, and (2) the correctness of the protocols PACKED-ADD and PACKED-MULT for addition gates and multiplication gates respectively.

We now describe the construction of the simulator $\mathcal{S}$.

1. In Step 1, $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{prep}}$ and receives the shares of corrupted parties.
2. In Step 2, $\mathcal{S}$ follows the protocol.
3. In Step 3, for every group of k input gates of Client_i, if Client_i is honest, $\mathcal{S}$ samples random elements as the shares of corrupted parties. If Client_i is corrupted, $\mathcal{S}$ learns the inputs $\boldsymbol{x}$ and the shares of corrupted parties (since $\mathcal{S}$ can access to the inputs and random tapes of corrupted clients and corrupted parties). Note that in both cases, $\mathcal{S}$ learns the shares of corrupted parties.
4. In Step 4.(a), $\mathcal{S}$ simulates NETWORK. Note that NETWORK only involves local computation and an invocation of $\mathcal{F}_{\text{tran}}$. $\mathcal{S}$ follows the protocol in NETWORK and computes the shares of corrupted parties. Then $\mathcal{S}$ emulates $\mathcal{F}_{\text{tran}}$ and receives the shares of corrupted parties. At the end of NETWORK, $\mathcal{S}$ learns the shares of $[\boldsymbol{x}]_{n-k}$ and $[\boldsymbol{y}]_{n-k}$ held by corrupted parties.

   In Step 4.(b), for each group of addition gates with input sharings $[\boldsymbol{x}]_{n-k}$ and $[\boldsymbol{y}]_{n-k}$, $\mathcal{S}$ simulates PACKED-ADD. Note that PACKED-ADD only involves local computation and an invocation of $\mathcal{F}_{\text{tran}}$. $\mathcal{S}$ follows the protocol in PACKED-ADD and computes the shares of corrupted parties. Then $\mathcal{S}$ emulates $\mathcal{F}_{\text{tran}}$ and receives the shares of corrupted parties. At the end of PACKED-ADD, $\mathcal{S}$ learns the shares of $[\boldsymbol{z}]_{n-k}$ held by corrupted parties.

   In Step 4.(b), for each group of multiplication gates with input sharings $[\boldsymbol{x}]_{n-k}$ and $[\boldsymbol{y}]_{n-k}$, $\mathcal{S}$ simulates PACKED-MULT as follows:

Figure 12.5: Protocol PACKED-MAIN

1. **Preprocessing Phase**. All parties invoke $\mathcal{F}_{\text{prep}}$ to prepare enough packed Beaver triples in the form of $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-k})$, where $\boldsymbol{a}, \boldsymbol{b}$ are random vectors in $\mathbb{F}^k$ and $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$.
2. **Initialization**. Let $\alpha_1, \alpha_2, \ldots, \alpha_n$ be distinct field elements in $\mathbb{F}$ which are used for the shares of all parties in packed Shamir secret sharing schemes. Let $\beta_1, \beta_2, \ldots, \beta_{|C|}$ be $|C|$ distinct field elements that are different from $\alpha_1, \alpha_2, \ldots, \alpha_n$. Let $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_k)$ be the default positions for the packed Shamir secret sharing schemes. We associate the field element β_i with the i-th gate in C.
3. **Input Phase**. Let $\mathsf{Client}_1, \mathsf{Client}_2, \ldots, \mathsf{Client}_{\mathsf{c}}$ denote the clients who provide inputs. All input gates are divided into groups of size k based on the input holders. For every group of k input gates of Client_i, suppose $\boldsymbol{x}$ are the inputs, and $\mathtt{pos} = (p_1, p_2, \ldots, p_k)$ are the positions associated with these k gates. Client_i generates a random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{x} \| \mathtt{pos}]_{n-k}$ and distributes the shares to all parties.
4. **Evaluation Phase**. All parties evaluate the circuit layer by layer as follows:
   (a) For the current layer, all gates are divided into groups of size k based on their types (i.e., multiplication gates or addition gates). For each group of k gates, let $\boldsymbol{x}$ be the first inputs of all gates, and $\boldsymbol{y}$ the second inputs of all gates. All parties invoke NETWORK to prepare $[\boldsymbol{x}]_{n-k}$ and $[\boldsymbol{y}]_{n-k}$.
   (b) For each group of k gates, let pos be the positions associated with these k gates.
      - If they are addition gates, all parties invoke PACKED-ADD on $([\boldsymbol{x}]_{n-k}, [\boldsymbol{y}]_{n-k})$, and obtain $[\boldsymbol{z} \| \mathtt{pos}]_{n-k}$.
      - If they are multiplication gates, all parties invoke PACKED-MULT on $([\boldsymbol{x}]_{n-k}, [\boldsymbol{y}]_{n-k})$ with a fresh packed Beaver triple $([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-k})$, and obtain $[\boldsymbol{z} \| \mathtt{pos}]_{n-k}$.
5. **Output Phase**. All output gates are divided into groups of size k based on the output receiver. For every group of k output gates of Client_i, suppose $\boldsymbol{x}$ are the inputs. All parties invoke the protocol NETWORK to prepare $[\boldsymbol{x}]_{n-k}$. Then, all parties send their shares to Client_i to let him reconstruct the result $\boldsymbol{x}$.

(a) The protocol PACKED-MULT consumes a fresh random packed Beaver triple

$$([\boldsymbol{a}]_{n-k}, [\boldsymbol{b}]_{n-k}, [\boldsymbol{c}]_{n-k}).$$

Recall that $\mathcal{S}$ learns the shares of these sharings held by corrupted parties when emulating the functionality $\mathcal{F}_{\text{prep}}$.

(b) In Step 2 of PACKED-MULT, $\mathcal{S}$ computes the shares of $[\boldsymbol{x}+\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{n-k}$ held by corrupted parties. Then $\mathcal{S}$ generates two random degree-$(n-k)$ packed Shamir sharings as $[\boldsymbol{x}+\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{n-k}$ based on the shares of corrupted parties.

(c) Since the whole sharings $[\boldsymbol{x}+\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{n-k}$ have been generated by $\mathcal{S}$, $\mathcal{S}$ honestly follows the protocol in Step 3 of PACKED-MULT. $\mathcal{S}$ learns the shares of $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{k-1}$ of corrupted parties at the end of this step.

(d) In Step 4 of PACKED-MULT, $\mathcal{S}$ computes the shares of $[\boldsymbol{z}]_{n-1}$ held by corrupted parties.

(e) In Step 5 of PACKED-MULT, $\mathcal{S}$ emulates $\mathcal{F}_{\text{tran}}$ and receives the shares of corrupted parties. At the end of this step, $\mathcal{S}$ learns the shares of $[\boldsymbol{z}]_{n-k}$ held by corrupted parties.

5. In Step 5, $\mathcal{S}$ simulates NETWORK in the same way as that for Step 4.(a). $\mathcal{S}$ invokes $\mathcal{F}_{\text{main}}$ with the input of corrupted clients, and receives the output of corrupted clients. For each group of output gates of Client_i, $\mathcal{S}$ has learned the shares of the input sharing $[\boldsymbol{x}]_{n-k}$ held by corrupted parties. If Client_i is honest, $\mathcal{S}$ does nothing. If Client_i is corrupted, $\mathcal{S}$ also learns the secret $\boldsymbol{x}$. $\mathcal{S}$ uses $\boldsymbol{x}$ and the shares of $[\boldsymbol{x}]_{n-k}$ held by corrupted parties, which are together $k+(n-t)=n-k+1$ values (recall that $t=n-2k+1$), to reconstruct the whole sharing $[\boldsymbol{x}]_{n-k}$ and finally sends the shares of $[\boldsymbol{x}]_{n-k}$ of honest parties to Client_i.

This completes the description of $\mathcal{S}$.

We show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. It is sufficient to focus on the places where honest parties and clients need to communicate with corrupted parties and clients:

- In Step 3, honest clients need to share its input. In the real world, an honest client will generate a random degree-$(n-k)$ packed Shamir sharing and distribute the shares to other parties. By the property of the degree-$(n-k)$ packed Shamir secret sharing scheme, the shares of corrupted parties are uniformly random. Therefore $\mathcal{S}$ perfectly simulates the behaviors of honest clients.
- In Step 4.(b), honest parties need to communicate with corrupted parties when running PACKED-MULT. In Step 3 of PACKED-MULT, all parties need to send their shares of $[\boldsymbol{x}+\boldsymbol{a}]_{n-k}, [\boldsymbol{y}+\boldsymbol{b}]_{n-k}$ to P_1. And P_i needs to distribute $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{k-1}$ to all parties.

  Note that $[\boldsymbol{x}+\boldsymbol{a}]_{n-k}=[\boldsymbol{x}]_{n-k}+[\boldsymbol{a}]_{n-k}$, and $[\boldsymbol{a}]_{n-k}$ is a random degree-$(n-k)$ packed Shamir sharing given the shares of corrupted parties. Therefore $[\boldsymbol{x}+\boldsymbol{a}]_{n-k}$ is also a random degree-$(n-k)$ packed Shamir sharing given the shares of corrupted parties. Similarly, $[\boldsymbol{y}+\boldsymbol{b}]_{n-k}$ is a random degree-$(n-k)$ packed Shamir sharing given the shares of corrupted parties. Also note that $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{k-1}$ are fully determined by the secrets $\boldsymbol{x}+\boldsymbol{a}$ and $\boldsymbol{y}+\boldsymbol{b}$.

  In the ideal world, $\mathcal{S}$ generates two random degree-$(n-k)$ packed Shamir sharings as $[\boldsymbol{x}+\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{n-k}$ based on the shares of corrupted parties. Then, the distribution of these two random degree-$(n-k)$ packed Shamir sharings is identical to that in the real world. Note that it also implies that the distribution of the two secrets $\boldsymbol{x}+\boldsymbol{a}$ and $\boldsymbol{y}+\boldsymbol{b}$ is identical to that in the real world. After sampling $[\boldsymbol{x}+\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{n-k}$, $\mathcal{S}$ honestly follows Step 3 of PACKED-MULT. Thus, $\mathcal{S}$ perfectly simulates the behaviors of honest parties.
- Finally, in Step 5, for each group of output gates of a corrupted client, honest parties need to send their shares of the sharing associated with these gates to corrupted clients. Note

that a degree-$(n-k)$ packed Shamir sharing is determined by its secret and the shares of corrupted parties. Since $\mathcal{S}$ learns the output of corrupted clients from $\mathcal{F}_{\text{main}}$ and the shares of corrupted parties, $\mathcal{S}$ can compute the shares of honest parties and perfectly simulate the behaviors of honest parties.

□

Analysis of the Communication Complexity. We assume that the number of multiplication gates is the same as the number of addition gates. We also assume that the number of input gates and output gates is much smaller than the number of addition gates and multiplication gates. Let C denote the circuit and DEPTH denote the circuit depth. We use I to denote the input size, G to denote the number of gates, and O to denote the output size. Then $|C| \geq I + G + O$.

- Cost of the Circuit Independent Preprocessing Phase: The size of the preprocessing data per packed Beaver triple is $3n$ field elements. Therefore, the total size of the preprocessing data for multiplication gates is $(\frac{G}{2k} + \mathsf{Depth}) \cdot 3n$. Here $\frac{G}{2}$ is the estimated number of multiplication gates, and $\frac{G}{2k} + \mathsf{Depth}$ is the number of packed Beaver triples required for multiplication gates. The reason of adding Depth is because the multiplication gates are grouped layer by layer. In each layer, if there are m multiplication gates, then we need to prepare $\lceil m/k \rceil < m/k + 1$ packed Beaver triples.
- Cost of the Online Phase:
  - Input Phase: For every group of k input gates, the communication complexity is n field elements. Let I denote the size of input. Then the cost of the input phase is $(\frac{I}{k} + \mathsf{c}) \cdot n$ field elements of communication. The reason of adding c is because the input gates are grouped based on the input holders.
  - Evaluation Phase: For every group of k (addition or multiplication) gates, all parties invoke NETWORK to prepare the two input sharings. Each invocation of NETWORK involves one invocation of $\mathcal{F}_{\text{tran}}$.

    For addition gates, each invocation of PACKED-ADD involves one invocation of $\mathcal{F}_{\text{tran}}$.

    For multiplication gates, each invocation of PACKED-MULT involves $4n$ elements of communication and one invocation of $\mathcal{F}_{\text{tran}}$.

    Thus, the evaluation phase cost $(\frac{G}{2k} + \mathsf{Depth}) \cdot 4n$ elements of communication and $(\frac{G}{k} + 2 \cdot \mathsf{Depth}) \cdot 3$ invocations of $\mathcal{F}_{\text{tran}}$. When using TRAN to instantiate $\mathcal{F}_{\text{tran}}$, $(\frac{G}{k} + 2 \cdot \mathsf{Depth}) \cdot 3$ invocations of $\mathcal{F}_{\text{tran}}$ requires $12(\frac{G}{k} + 2 \cdot \mathsf{Depth}) \cdot \frac{n^2}{k}$ field elements of preprocessing data, and $12(\frac{G}{k} + 2 \cdot \mathsf{Depth}) \cdot n$ field elements of communication.

    In total, the cost of the evaluation phase is $12(\frac{G}{k} + 2 \cdot \mathsf{Depth}) \cdot \frac{n^2}{k}$ field elements of preprocessing data, and $14(\frac{G}{k} + 2 \cdot \mathsf{Depth}) \cdot n$ field elements of communication.
  - Output Phase: For every group of k output gates, all parties invoke NETWORK to prepare the input sharing. Then all parties send their shares to the client who should receive the result. Recall that each invocation of NETWORK involves one invocation of $\mathcal{F}_{\text{tran}}$. Thus, the output phase cost $(\frac{O}{k} + \mathsf{c}) \cdot n$ field elements of communication and $\frac{O}{k} + \mathsf{c}$ invocations of $\mathcal{F}_{\text{tran}}$.

With a similar analysis, the cost of the output phase is $4(\frac{O}{k} + \mathsf{c}) \cdot \frac{n^2}{k}$ field elements of preprocessing data, and $5(\frac{O}{k} + \mathsf{c}) \cdot n$ field elements of communication.

In summary, the total cost of the online phase is $(\frac{12G}{k} + \frac{4O}{k} + 24 \cdot \mathsf{Depth} + 4\mathsf{c}) \cdot \frac{n^2}{k}$ elements of preprocessing data, and $(\frac{I}{k} + \frac{14G}{k} + \frac{5O}{k} + 28 \cdot \mathsf{Depth} + 6\mathsf{c}) \cdot n$ elements of communication.

Thus, our semi-honest protocol requires $12|C| \cdot \frac{n^2}{k^2} + O((\mathsf{Depth} + \mathsf{c}) \cdot \frac{n^2}{k})$ elements of preprocessing data, and $14|C| \cdot \frac{n}{k} + O((\mathsf{Depth} + \mathsf{c}) \cdot n)$ elements of communication.

Theorem 12.1. *In the client-server model, let* c *denote the number of clients,* n *denote the number of parties (servers), and* t *denote the number of corrupted parties (servers). Let* $\mathbb{F}$ *be a finite field of size* $|\mathbb{F}| \geq |C| + n$*. For an arithmetic circuit* C *over* $\mathbb{F}$*, there exists an information-theoretic MPC protocol in the preprocessing model which securely computes the arithmetic circuit* C *in the presence of a semi-honest adversary controlling up to* c *clients and* t *parties. The cost of the protocol is* $O(|C| \cdot \frac{n^2}{k^2} + (\mathsf{Depth} + \mathsf{c}) \cdot \frac{n^2}{k})$ *field elements of preprocessing data and* $O(|C| \cdot \frac{n}{k} + (\mathsf{Depth} + \mathsf{c}) \cdot n)$ *field elements of communication, where* $k = \frac{n-t+1}{2}$ *and* Depth *is the circuit depth.*

Chapter 13

An Alternative Approach for Network Routing

In this chapter, we discuss an alternative approach for the network routing which works for small fields $|\mathbb{F}| \geq 2n$.

13.1 Preliminaries: Hall's Marriage Theorem

Definition 13.1 (Bipartite Graph)**.** *A graph $G = (V, E)$ is a bipartite graph if there exists a partition (V_1, V_2) of V such that for all edge $(v_i, v_j) \in E$, $v_i \in V_1$ and $v_j \in V_2$.*

In the following, we will use (V_1, V_2, E) to denote a bipartite graph. We say a bipartite graph (V_1, V_2, E) is d-regular if the degree of each vertex in $V_1 \bigcup V_2$ is d.

Definition 13.2 (Perfect Matching)**.** *For a bipartite graph (V_1, V_2, E) such that $|V_1| = |V_2|$, a perfect matching is a subset of edges $\mathcal{E} \in E$ which satisfies that each vertex in the sub-graph $(V_1, V_2, \mathcal{E})$ has degree 1.*

Theorem 10.2 (Hall's Marriage Theorem)**.** *For a bipartite graph (V_1, V_2, E) such that $|V_1| = |V_2|$, there exists a perfect matching iff for all subset $V_1' \subset V_1$, the number of the neighbors of vertices in V_2 is at least $|V_1'|$.*

In this work, we will make use of the following two well-known corollaries of Hall's Marriage Theorem. For completeness, we also provide proofs for the corollaries.

Corollary 10.1. *There exists a perfect matching in a d-regular bipartite graph.*

Proof. Let $G = (V_1, V_2, E)$ denote the d-regular bipartite graph. To show the existence of a perfect matching in G, it is sufficient to examine the requirement of Hall's Marriage Theorem.

For all subset $V_1' \subset V_1$, let $N(V_1')$ denote the set of vertices in V_2 which are connected to vertices in V_1'. It is sufficient to show that $|N(V_1')| \geq |V_1'|$. Consider the sub-graph $G' = (V_1', N(V_1'), E')$ which contains all the edges between V_1' and $N(V_1')$. Since G is a d-regular graph, the number of edges in G' is $|E'| = d \cdot |V_1'|$. On the other hand, we have $d \cdot |N(V_1')| \geq |E'|$ since the degree of each vertex in $|N(V_1')|$ is upper bounded by d. Therefore $d \cdot |N(V_1')| \geq |E'| = d \cdot |V_1'|$, which means $|N(V_1')| \geq |V_1'|$. □

13.2 Network Routing using Hall's Marriage Theorem

As discussed in Chapter 10.2.4, we will always use the default positions for the packed Shamir secret sharing scheme in our construction. This means that

- For Protocol PACKED-ADD (Protocol 12.3), there is no need to run the last step and the protocol only contains local computation.
- For Protocol PACKED-MULT (Protocol 12.4), the last step becomes a degree reduction.

Recall that our idea is to realize the non-collision property:

Property 10.1 (Non-collision)**.** *For each input sharing of each layer, the secrets of this input sharing come from different positions in the output sharings of previous layers.*

To this end, for each (input, addition, or multiplication) gate, we copy the output wire the number of times it is used in later layers.

Evaluating Fan-Out Gates. Let $[\boldsymbol{x}]_{n-k}$ denote the output packed Shamir sharing of a group of (input, addition, or multiplication) gates, and let n_i denote the number of times that the i-th secret x_i is used in later layers for all $i \in \{1, 2, \ldots, k\}$. We first transform $\boldsymbol{x}$ to $m = \lceil \frac{n_1+n_2+\ldots+n_k}{k} \rceil$ vectors $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}$ in $\mathbb{F}^k$ such that they contain n_i copies of the value x_i for all $i \in \{1, 2, \ldots, k\}$. All parties use the following algorithm to locally determine what values should be in each of these m vectors.

1. All parties locally initiate an empty list L. From $i = 1$ to k, all parties locally insert n_i times of x_i into L.
2. From $i = 1$ to m, all parties locally set $\boldsymbol{x}^{(i)}$ to be the vector of the first k elements in L, and then remove these elements from L. For the last vector, we insert enough number of 0s if the number of elements in L is smaller than k. In this way, all parties determine the values that should be in the m vectors $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}$.

For each $\boldsymbol{x}^{(i)} \in \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$, to obtain $[\boldsymbol{x}^{(i)}]_{n-k}$ from $[\boldsymbol{x}]_{n-k}$, all parties set $\mathcal{L}'_i$ to be the $\mathbb{F}$-linear map that maps $\boldsymbol{x}$ to $\boldsymbol{x}^{(i)}$, and then perform a sharing transformation to transform $[\boldsymbol{x}]_{n-k}$ to $[L'_i(\boldsymbol{x})]_{n-k}$.

Achieving Non-Collision Properties. After performing the fan-out gates, there is a bijective map between the output wires (of the input layer and all intermediate layers) and the input wires (of the output layer and all intermediate layers).

In the following, when we use the term "output sharings", we refer to the output sharings from the input layer and all intermediate layers. When we use the term "input sharings", we refer to the input sharings of the output layer and all intermediate layers.

Since every output wire from every layer is only used once as an input wire of another layer, the number of output sharings in the circuit is the same as the number of input sharings in the circuit. Let m denote the number of output sharings in the circuit. Then the number of input sharings is also m. We will label all the output sharings by $1, 2, \ldots, m$ and all the input sharings also by $1, 2, \ldots, m$. Consider a matrix $\boldsymbol{N} \in \{1, 2, \ldots, m\}^{m \times k}$ where $\boldsymbol{N}_{i,j}$ is the index of the input sharing that the j-th secret of the i-th output sharing wants to go to. Then for all $\ell \in \{1, 2, \ldots, m\}$, there are exactly k entries of $\boldsymbol{N}$ which are equal to ℓ. And the secrets at those

positions are the secrets we want to collect for the ℓ-th input sharing. We will prove the following theorem.

Theorem 10.1. *Let $m \geq 1, k \geq 1$ be integers. Let $\boldsymbol{N}$ be a matrix of dimension $m \times k$ in $\{1, 2, \ldots, m\}^{m \times k}$ such that for all $\ell \in \{1, 2, \ldots, m\}$, the number of entries of $\boldsymbol{N}$ which are equal to ℓ is k. Then, there exists m permutations $p_1, p_2, \ldots, p_m$ over $\{1, 2, \ldots, k\}$ such that after performing the permutation p_i on the i-th row of $\boldsymbol{N}$, the new matrix $\boldsymbol{N}'$ satisfies that each column of $\boldsymbol{N}'$ is a permutation over $(1, 2, \ldots, m)$. Furthermore, the permutations $p_1, p_2, \ldots, p_m$ can be found within polynomial time.*

Jumping ahead, when we apply p_i to the i-th output sharing for all $i \in \{1, 2, \ldots, m\}$, Theorem 10.1 guarantees that for all $j \in \{1, 2, \ldots, k\}$ the j-th secrets of all output sharings need to go to different input sharings. Note that this ensures the non-collision property. During the computation, we will perform the permutation p_i on the i-th output sharing right after it is computed. Note that when preparing an input sharing, the secrets we need only come from the output sharings which have been computed. The secrets of these output sharings have been properly permuted such that the secrets we want are in different positions. Therefore, we can use $\mathcal{F}_{\text{select-semi}}$ to choose these secrets and obtain the desired input sharing. With the non-collision property, we can achieve the network routing by following a similar approach to that in Chapter 12.2.2.

Proof of Theorem 10.1. We will prove the theorem by induction on k.

When $k = 1$, $\boldsymbol{N}$ is a matrix of dimension $m \times 1$. Since each value $\ell \in \{1, 2, \ldots, m\}$ appears once in $\boldsymbol{N}$, and $\boldsymbol{N}$ only has one column, the column of $\boldsymbol{N}$ is a permutation of $(1, 2, \ldots m)$. The permutations $p_1, p_2, \ldots, p_m$ will just be the identities.

Assume the theorem holds for $k = k' \geq 1$. Let's consider the case when $k = k' + 1$. We first construct a bipartite graph $G = (V_1, V_2, E)$. Let $V_1 = V_2 = \{1, 2, \ldots, m\}$. For all $i \in \{1, 2, \ldots, m\}$ and $j \in \{1, 2, \ldots, k\}$, if $\boldsymbol{N}_{i,j} = \ell$, we create an edge $(i, \ell) \in V_1 \times V_2$ and insert (i, ℓ) in E.

Now we show that G is a regular-k bipartite graph. For all vertex $i \in V_1$, we create an edge for each entry in the i-th row of $\boldsymbol{N}$. Therefore, the degree of the vertex i is k. For all vertex $\ell \in V_2$, we create an edge for each entry in $\boldsymbol{N}$ *which is equal to* ℓ. Since in $\boldsymbol{N}$, the number of entries that are equal to ℓ is k, the degree of each vertex $\ell \in V_2$ is k. Thus, G is a regular-k bipartite graph.

According to Corollary 10.1, there exists a perfect matching in G. Suppose $(1, \ell_1), \ldots, (m, \ell_m)$ are the edges in the perfect matching. Then $(\ell_1, \ell_2, \ldots, \ell_m)$ is a permutation of $(1, 2, \ldots, m)$. For all $i \in \{1, 2, \ldots, m\}$, let j_i be the first position that $\boldsymbol{N}_{i,j_i} = \ell_i$. Now for all $i \in \{1, 2, \ldots, m\}$, we switch the k-th entry and the j_i-th entry in the i-th row, and denote the new matrix to be $\tilde{\boldsymbol{N}}$. In this way, the last column of $\tilde{\boldsymbol{N}}$ becomes a permutation of $(1, 2, \ldots, m)$.

Let $\boldsymbol{M}$ be the sub-matrix of $\tilde{\boldsymbol{N}}$ which contains the first $k-1$ columns. Then $\boldsymbol{M}$ is a matrix in $\{1, 2, \ldots, m\}^{m \times (k-1)}$, and for all $\ell \in \{1, 2, \ldots, m\}$, the number of entries of $\boldsymbol{M}$ which are equal to ℓ is $k - 1$. According to the induction hypothesis, there exists m permutations $p'_1, p'_2, \ldots, p'_m$ over $\{1, 2, \ldots, k-1\}$ such that after performing the permutation p'_i on the i-th row of $\boldsymbol{M}$, the new matrix $\boldsymbol{M}'$ satisfies that each column of $\boldsymbol{M}'$ is a permutation of $(1, 2, \ldots, m)$.

Now we construct the permutations $p_1, p_2, \ldots, p_m$ over $\{1, 2, \ldots, k\}$ as follows: For all $i \in \{1, 2, \ldots, m\}$

- for all $j \in \{1, 2, \ldots, k-1\}$ and $j \neq j_i$, $p_i(j) = p'_i(j)$;
- for j_i, $p_i(j_i) = k$;
- for k, $p_i(k) = p'_i(j_i)$.

Let $\boldsymbol{N}'$ denote the matrix after performing the permutation p_i on the i-th row of $\boldsymbol{N}$ for all $i \in \{1, 2, \ldots, m\}$. Note that for all $i \in \{1, 2, \ldots, m\}$, performing the permutation p_i on the i-th row of $\boldsymbol{N}$ is equivalent to first switching the k-th entry and the j_i-th entry and then performing the permutation p'_i on the first $k-1$ entries. Therefore, for all $j \in \{1, 2, \ldots, k-1\}$, the j-th column of $\boldsymbol{N}'$ is the same as the j-th column of $\boldsymbol{M}'$, and the k-th column of $\boldsymbol{N}'$ is the same as the k-th column of $\tilde{\boldsymbol{N}}$. Therefore each column of $\boldsymbol{N}'$ is a permutation of $(1, 2, \ldots, m)$.

According to Corollary 10.1, the perfect matching of G can be found within polynomial time. According to the induction hypothesis, $p'_1, p'_2, \ldots, p'_m$ can be found within polynomial time. Therefore, $p_1, p_2, \ldots, p_m$ can be constructed within polynomial time.

Therefore, the theorem holds for all integers $k \geq 1$. □

Thus, to achieve the non-collision property, we perform the permutation computed from the algorithm in Theorem 10.1 on $\boldsymbol{x}^{(i)}$. Let $\mathcal{L}''_i$ denote the linear map that corresponds to this permutation. Then, all parties can achieve the non-collision property by performing a sharing transformation from $[\boldsymbol{x}^{(i)}]_{n-k}$ to $[\mathcal{L}''_i(\boldsymbol{x}^{(i)})]_{n-k}$.

We note that all parties can obtain $[\mathcal{L}''_i(\boldsymbol{x}^{(i)})]_{n-k}$ directly from $[\boldsymbol{x}]_{n-k}$: Since $\mathcal{L}'_i, \mathcal{L}''_i$ are linear maps, $\mathcal{L}_i = \mathcal{L}'_i \circ \mathcal{L}''_i$ is also a linear map. Thus, all parties can perform a single sharing transformation from $[\boldsymbol{x}]_{n-k}$ to $[\mathcal{L}_i(\boldsymbol{x})]_{n-k}$.

Preparing Desired Input Sharings. Suppose $[\boldsymbol{x}^{(1)}]_{n-k}, \ldots, [\boldsymbol{x}^{(k)}]_{n-k}$ are k degree-$(n-k)$ packed Shamir sharings from previous layers (which do not need to be distinct) and all parties want to prepare a degree-t packed Shamir sharing $[\boldsymbol{x}]_{n-k}$ that contains the i_j-th secret of $\boldsymbol{x}^{(j)}$ for all $j \in \{1, 2, \ldots, k\}$. According to the non-collision property, $i_1, i_2, \ldots, i_k$ are all distinct. All parties locally compute a degree-$(n-1)$ packed Shamir sharing that contains the secrets we want as follows:

1. For all $j \in \{1, 2, \ldots, k\}$, let $\boldsymbol{e}_j$ denote a vector in $\mathbb{F}^k$ where all entries of $\boldsymbol{e}_j$ are 0 except the i_j-th entry is 1. All parties locally transform $\boldsymbol{e}_j$ to a degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{e}_j]_{k-1}$.
2. All parties locally compute

$$[\boldsymbol{x}']_{n-1} = \sum_{j=1}^{k} [\boldsymbol{e}_j]_{k-1} \cdot [\boldsymbol{x}^{(j)}]_{n-k}.$$

The correctness follows from the facts that $[\boldsymbol{e}_j]_{k-1} \cdot [\boldsymbol{x}^{(j)}]_{n-k} = [\boldsymbol{e}_j * \boldsymbol{x}^{(j)}]_{n-1}$ and $\boldsymbol{e}_j * \boldsymbol{x}^{(j)}$ is a vector in $\mathbb{F}^k$ where all entries are 0 except the i_j-th entry is $x^{(j)}_{i_j}$.

To obtain $[\boldsymbol{x}]_{n-k}$ from $[\boldsymbol{x}']_{n-1}$, let L denote the linear map that maps $\boldsymbol{x}'$ to $\boldsymbol{x}$, all parties perform a sharing transformation from $[\boldsymbol{x}']_{n-1}$ to $[\boldsymbol{x}]_{n-k}$.

Summary. We summarize the alternative approach for the network routing in Protocol NETWORK-HALL-PART I and NETWORK-HALL-PART II (described in Protocol 13.1 and Protocol 13.2 respectively).

Figure 13.1: Protocol NETWORK-HALL-1

1. **Circuit Preprocessing**:
   - For each layer of the circuit, all gates are divided into groups of size k based on their types. For every group of k (input, addition, or multiplication) gates, let $\boldsymbol{x}$ denote the values associated with the output wires of these k gates. Let n_i denote the number of times that the i-th secret x_i is used in later layers for all $i \in \{1, 2, \ldots, k\}$. All parties transform $\boldsymbol{x}$ to $m = \lceil \frac{n_1+n_2+\ldots+n_k}{k} \rceil$ vectors $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}$ in $\mathbb{F}^k$ such that they contain n_i copies of the value x_i for all $i \in \{1, 2, \ldots, k\}$. All parties use the following algorithm to locally determine what values should be in each of these m vectors.
     (a) All parties locally initiate an empty list List. From $i = 1$ to k, all parties locally insert n_i times of x_i into List.
     (b) From $i = 1$ to m, all parties locally set $\boldsymbol{x}^{(i)}$ to be the vector of the first k elements in List, and then remove these elements from List. For the last vector, we insert enough number of 0s if the number of elements in List is smaller than k.
   - Let N denote the number of output sharings of the input layer and all intermediate layers. Then the number of input sharings of the output layer and all intermediate layers is also N. The output sharings are labeled by $1, 2, \ldots, N$, and the input sharings are also labeled by $1, 2, \ldots, N$. All parties construct a matrix $\boldsymbol{M} \in \{1, 2, \ldots, N\}^{N \times k}$ where $\boldsymbol{M}_{i,j}$ is the index of the input sharing that the j-th secret in the i-th output sharing wants to go. All parties use a deterministic algorithm that all parties agree on to compute N permutations $p_1, p_2, \ldots, p_N$ such that after applying p_i to the i-th row of $\boldsymbol{M}$, the new matrix $\boldsymbol{M}'$ satisfies that each column of $\boldsymbol{M}'$ is a permutation of $(1, 2, \ldots, N)$. The existence of such an algorithm is guaranteed by Theorem 10.1. Then, all parties associate p_i with the i-th output sharing.

Now we can replace the protocol NETWORK by NETWORK-HALL. In particular, the protocol works for any finite field $\mathbb{F}$ of size $|\mathbb{F}| \geq 2n$.

Theorem 13.1. *In the client-server model, let* c *denote the number of clients,* n *denote the number of parties (servers), and* t *denote the number of corrupted parties (servers). Let* $\mathbb{F}$ *be a finite field of size* $|\mathbb{F}| \geq 2n$*. For an arithmetic circuit* C *over* $\mathbb{F}$*, there exists an information-theoretic MPC protocol in the preprocessing model which securely computes the arithmetic circuit* C *in the presence of a semi-honest adversary controlling up to* c *clients and* t *parties. The cost of the protocol is* $O(|C| \cdot \frac{n^2}{k^2} + (\mathsf{Depth} + \mathsf{c}) \cdot \frac{n^2}{k})$ *field elements of preprocessing data and*

Figure 13.2: Protocol NETWORK-HALL-2

2. **Achieving the Non-Collision Property**: For every group of k (input, addition, or multiplication) gates, let $\boldsymbol{x}$ denote the values associated with the output wires of these k gates. After all parties compute $[\boldsymbol{x}]_{n-k}$, all parties does the following:
   (a) Recall the vectors $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}$ computed in the last step. For all $i \in \{1, 2, \ldots, m\}$, let L'_i denote the linear map that maps $\boldsymbol{x}$ to $\boldsymbol{x}^{(i)}$. Also recall the permutation associated with each vector $\boldsymbol{x}^{(i)}$ computed in the last step. Let L''_i denote the linear map that corresponds to the permutation associated with $\boldsymbol{x}^{(i)}$.
   (b) For all $i \in \{1, 2, \ldots, m\}$, let $L_i = L''_i \circ L'_i$. Let Σ be the degree-$(n-1)$ packed Shamir secret sharing scheme. Let Σ' be the degree-$(n-k)$ packed Shamir secret sharing scheme.

   All parties invoke $\mathcal{F}_{\text{tran}}$ with (Σ, Σ', L_i) and $[\boldsymbol{x}]_{n-k}$, and output $[L_i(\boldsymbol{x})]_{n-k}$.
3. **Preparing Desired Input Sharings**: Suppose all parties want to prepare a degree-$(n-k)$ packed Shamir sharing of $\boldsymbol{x}$. Suppose $[\boldsymbol{x}^{(1)}]_{n-k}, \ldots, [\boldsymbol{x}^{(k)}]_{n-k}$ are k degree-$(n-k)$ packed Shamir sharings from previous layers (which do not need to be distinct) such that $\boldsymbol{x}^{(j)}$ contains the value x_j at position i_j for all $j \in \{1, 2, \ldots, k\}$. According to the non-collision property, $i_1, i_2, \ldots, i_k$ are all distinct.
   (a) For all $j \in \{1, 2, \ldots, k\}$, let $\boldsymbol{e}_{i_j}$ denote a vector in $\mathbb{F}^k$ where all entries of $\boldsymbol{e}_{i_j}$ are 0 except the i_j-th entry is 1. All parties locally transform $\boldsymbol{e}_{i_j}$ to a degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{e}_{i_j}]_{k-1}$.
   (b) All parties locally compute

$$[\boldsymbol{x}']_{n-1} = \sum_{j=1}^{k} [\boldsymbol{e}_{i_j}]_{k-1} \cdot [\boldsymbol{x}^{(j)}]_{n-k}.$$

   (c) Let $f : \mathbb{F}^k \to \mathbb{F}^k$ be a linear map such that $\boldsymbol{x} = f(\boldsymbol{x}')$. Let Σ be the degree-$(n-1)$ packed Shamir secret sharing scheme. Let Σ' be the degree-$(n-k)$ packed Shamir secret sharing scheme.

   All parties invoke $\mathcal{F}_{\text{tran}}$ with (Σ, Σ', f) and $[\boldsymbol{x}']_{n-1}$, and output $[\boldsymbol{x}]_{n-k}$.

$O(|C| \cdot \frac{n}{k} + (\mathsf{Depth} + \mathsf{c}) \cdot n)$ field elements of communication, where $k = \frac{n-t+1}{2}$ and Depth is the circuit depth.

Chapter 14

Maliciously Secure Protocol

In this chapter, we discuss how to achieve malicious security. One difficulty is that corrupted parties can change the secret of a degree-$(n-k)$ packed Shamir sharing by changing their own shares locally. We extend the idea of using information-theoretic MACs introduced in [11, 30] to authenticate packed Shamir sharings.

Concretely, all parties will prepare a random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{\gamma}]_{n-k}$, where $\boldsymbol{\gamma} = (\gamma, \gamma, \ldots, \gamma) \in \mathbb{F}^k$ and γ is a random field element in $\mathbb{F}$, in the preprocessing phase (using the default positions). During the computation, a degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{x} \| \mathtt{pos}]_{n-k}$ is authenticated by computing a degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{\gamma} * \boldsymbol{x} \| \mathtt{pos}]_{n-k}$. We use $[\![\boldsymbol{x} \| \mathtt{pos}]\!]_{n-k}$ to denote the pair $([\boldsymbol{x} \| \mathtt{pos}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{x} \| \mathtt{pos}]_{n-k})$. Note that if corrupted parties change the secret $\boldsymbol{x}$ to $\boldsymbol{x}'$, they also need to change the secret $\boldsymbol{\gamma} * \boldsymbol{x}$ to $\boldsymbol{\gamma} * \boldsymbol{x}'$. However, since $\boldsymbol{\gamma}$ is a uniform vector in $\mathbb{F}^k$, the probability of a success attack is at most $1/|\mathbb{F}|$. When the field size is large enough, we can detect such an attack with overwhelming probability. Therefore in the following, we assume $|\mathbb{F}| \geq 2^\kappa$, where κ is the security parameter.

Recall that t is the number of corrupted parties, and $k = (n-t+1)/2$. Then the number of honest parties is $n - t = 2k - 1$. We focus on $\frac{n-1}{3} \leq t \leq n-1$. Then k satisfies that $2k - 2 \leq n - k$. The benefit is that for any $2k-1$ field elements in $\mathbb{F}$, there is a degree-$(n-k)$ packed Shamir sharing such that the shares of honest parties are these $2k-1$ elements. In this way, we can transform any attack that adds errors to the shares of honest parties to an attack that adds errors to the secrets.

14.1 Performing Sharing Transformation with Malicious Security

We will only focus on the sharing transformation we use in our semi-honest protocol: For two (potentially different) vectors of field elements $\mathtt{pos} = (p_1, \ldots, p_k)$ and $\mathtt{pos}' = (p'_1, \ldots, p'_k)$ such that they are disjoint with $\{\alpha_1, \ldots, \alpha_n\}$, all parties start with a degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{x} \| \mathtt{pos}]_{n-1}$. They want to compute a degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{y} \| \mathtt{pos}']_{n-k}$ such that for all $i \in \{1, 2, \ldots, k\}$, y_i is equal to x_j for some j.

We first construct a protocol that allows all parties to prepare a pair of random sharings for the above transformation. It follows from Protocol RAND-SHARING with the concrete improvement

discussed in Chapter 11.5.

14.1.1 Preparing Random Sharings for Sharing Transformations

Let $\Pi = \Pi(\mathtt{pos}, \mathtt{pos}', f)$ be the $\mathbb{F}$-linear secret sharing scheme defined as follows:

- The secret space $Z = \mathbb{F}^k$.
- The share space $U = \mathbb{F}^2$.
- For a secret $\boldsymbol{x} \in Z$, the sharing of $\boldsymbol{x}$ is $([\boldsymbol{x} \| \mathtt{pos}]_{n-1}, [f(\boldsymbol{x}) \| \mathtt{pos}']_{n-k})$.
- For a sharing $\boldsymbol{X} = (\boldsymbol{X}_1, \boldsymbol{X}_2)$, we view $\boldsymbol{X}_1$ as a degree-$(n-1)$ packed Shamir secret sharing scheme and reconstruct the secret $\boldsymbol{x}$.

Our goal is to prepare a pair of random sharings $([\boldsymbol{r} \| \mathtt{pos}]_{n-1}, [f(\boldsymbol{r}) \| \mathtt{pos}']_{n-k})$ where $\boldsymbol{r}$ is a random vector in $\mathbb{F}^k$. In the malicious security setting, we allow an adversary to add a constant vector to the secret of the second sharing. We summarize the functionality $\mathcal{F}_{\text{rand-sharing-mal}}$ in Functionality 14.1.

Figure 14.1: Functionality $\mathcal{F}_{\text{rand-sharing-mal}}$

1. $\mathcal{F}_{\text{rand-sharing-mal}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$. $\mathcal{F}_{\text{rand-sharing-mal}}$ also receives $\Pi = \Pi(\mathtt{pos}, \mathtt{pos}', f)$.
2. $\mathcal{F}_{\text{rand-sharing-mal}}$ receives from the adversary a set of shares $\{(u_j, v_j)\}_{j \in \mathcal{C}orr}$, and a constant vector $\boldsymbol{d} \in \mathbb{F}^k$.
3. $\mathcal{F}_{\text{rand-sharing-mal}}$ samples a random vector $\boldsymbol{r} \in \mathbb{F}^k$ and computes $f(\boldsymbol{r}) + \boldsymbol{d}$.
4. $\mathcal{F}_{\text{rand-sharing-mal}}$ samples a pair of random sharings $([\boldsymbol{r} \| \mathtt{pos}]_{n-1}, [f(\boldsymbol{r}) + \boldsymbol{d} \| \mathtt{pos}']_{n-k})$ such that for all $P_j \in \mathcal{C}orr$, the j-th share of $([\boldsymbol{r} \| \mathtt{pos}]_{n-1}, [f(\boldsymbol{r}) + \boldsymbol{d} \| \mathtt{pos}']_{n-k})$ is (u_j, v_j).
5. $\mathcal{F}_{\text{rand-sharing-mal}}$ distributes the shares of $([\boldsymbol{r} \| \mathtt{pos}]_{n-1}, [f(\boldsymbol{r}) + \boldsymbol{d} \| \mathtt{pos}']_{n-k})$ to honest parties.

Let $\Pi_1, \Pi_2, \ldots, \Pi_k$ be k $\mathbb{F}$-linear secret sharing schemes in the above form. We will prepare k random sharings, one for each secret sharing scheme. Our protocol will use $\mathcal{F}_{\text{rand}}$ to prepare random degree-$(n-k)$ packed Shamir sharings, and $\mathcal{F}_{\text{randZero}}$ to prepare random degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$. The description of Protocol RAND-SHARING-MAL appears in Protocol 14.2.

Lemma 14.1. *For all $k \leq (n+2)/3$, Protocol* RAND-SHARING-MAL *securely computes $\mathcal{F}_{\text{rand-sharing-mal}}$ in the $\{\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{randZero}}\}$-hybrid model against a fully malicious adversary who controls $t = n - 2k + 1$ parties.*

Proof. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}orr$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties.

The simulator $\mathcal{S}$ works as follows.

Figure 14.2: Protocol RAND-SHARING-MAL

1. For all $i \in \{1, 2, \ldots, k\}$, let $\Pi_i = \Sigma(\mathtt{pos}_i, \mathtt{pos}'_i, f_i)$.
2. All parties invoke $\mathcal{F}_{\text{randZero}}$ and obtain $(2k-1)$ random degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$, $[\boldsymbol{o}_1]_{n-1}, \ldots, [\boldsymbol{o}_{2k-1}]_{n-1}$.
3. For all $j \in \{1, \ldots, n\}$, all parties invoke $\mathcal{F}_{\text{rand}}$ and obtain a random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{r}_j]_{n-k}$. Then all parties send their shares to P_j. P_j views it as a degree-$(n-1)$ packed Shamir sharing and reconstructs $\boldsymbol{r}_j$, which is used as his shares of the degree-$(n-1)$ packed Shamir sharings in $\Pi_1, \ldots, \Pi_k$.
4. For all $j \in \{1, \ldots, n-2k+1\}$, all parties invoke $\mathcal{F}_{\text{rand}}$ and obtain a random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{r}_{n+j}]_{n-k}$. Then all parties send their shares to P_j. P_j views it as a degree-$(n-1)$ packed Shamir sharing and reconstructs $\boldsymbol{r}_{n+j}$, which is used as his shares of the degree-$(n-k)$ packed Shamir sharings in $\Pi_1, \ldots, \Pi_k$.
5. For all $i \in \{1, \ldots, k\}$, the shares of the degree-$(n-1)$ packed Shamir sharing in Π_i, $[\boldsymbol{\tau}_i \| \mathtt{pos}_i]_{n-1}$, are set to be $(r_{1,i}, \ldots, r_{n,i})$. And the first $n-2k+1$ shares of the degree-$(n-k)$ packed Shamir sharing in Π_i, $[f_i(\boldsymbol{\tau}_i) \| \mathtt{pos}'_i]_{n-k}$, are set to be $(r_{n+1,i}, \ldots, r_{2n-2k+1,i})$.
6. Note that each value in $\boldsymbol{\tau}_i$ is a linear combination of $(r_{1,i}, \ldots, r_{n,i})$. Each value in $f_i(\boldsymbol{\tau}_i)$ is a linear combination of the values in $\boldsymbol{\tau}_i$. And for all $j \in \{n-2k+2, \ldots, n\}$, the j-th share of $[f_i(\boldsymbol{\tau}_i) \| \mathtt{pos}'_i]_{n-k}$ is a linear combination of its first $n-2k+1$ shares $(r_{n+1,i}, \ldots, r_{2n-2k+1,i})$ and the secret $f_i(\boldsymbol{\tau}_i)$. Thus, the j-th share of $[f_i(\boldsymbol{\tau}_i) \| \mathtt{pos}'_i]_{n-k}$ is a linear combination of $(r_{1,i}, \ldots, r_{2n-2k+1,i})$. Let $c^{(i)}_{j,1}, \ldots, c^{(j)}_{j,2n-2k+1}$ be the coefficient such that the j-th share of $[f_i(\boldsymbol{\tau}_i) \| \mathtt{pos}'_i]_{n-k}$

$$r_{n+j,i} = \sum_{v=1}^{2n-2k+1} c^{(i)}_{j,v} \cdot r_{v,i}.$$

   For all $j \in \{n-2k+2, \ldots, n\}$ and $v \in \{1, \ldots, 2n-2k+1\}$, let $\boldsymbol{c}_{j,v} = (c^{(1)}_{j,v}, \ldots, c^{(k)}_{j,v})$.
7. For all $j \in \{n-2k+2, \ldots, n\}$, all parties locally compute

$$[\boldsymbol{r}_{n+j}]_{n-1} = [\boldsymbol{o}_{j-n+2k-1}]_{n-1} + \sum_{v=1}^{2n-2k+1} \boldsymbol{c}_{j,v} \cdot [\boldsymbol{r}_v]_{n-k}.$$

   Then, all parties send their shares of $[\boldsymbol{r}_{n+j}]_{n-1}$ to P_j.
8. For all $j \in \{n-2k+2, \ldots, n\}$, P_j reconstructs $[\boldsymbol{r}_{n+j}]_{n-1}$ and learns $\boldsymbol{r}_{n+j}$. Then, for all $i \in \{1, 2, \ldots, k\}$, all parties output $[\boldsymbol{\tau}_i \| \mathtt{pos}_i]_{n-1} = (r_{1,i}, \ldots, r_{n,i})$ and $[f_i(\boldsymbol{\tau}_i) \| \mathtt{pos}'_i]_{n-k} = (r_{n+1,i}, \ldots, r_{2n,i})$.

1. In Step 2, $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{randZero}}$ and receives the shares of $[\boldsymbol{o}_1]_{n-1}, \ldots, [\boldsymbol{o}_{2k-1}]_{n-1}$ held by corrupted parties.
2. In Step 3, $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{rand}}$ and receives the shares of each $[\boldsymbol{r}_j]_{n-k}$ held by corrupted parties.
   - For all corrupted party P_j, $\mathcal{S}$ generates a random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{r}_j]_{n-k}$ based on the shares held by corrupted parties. Then $\mathcal{S}$ sends the shares held by honest parties to P_j.
   - For all honest party P_j, $\mathcal{S}$ receives from corrupted parties their shares of $[\boldsymbol{r}_j]_{n-k}$. Then $\mathcal{S}$ sets a degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\delta}_j]_{n-1}$ as follows:
     (a) The share of each honest party is set to be 0.
     (b) The share of each corrupted party is set to be the share of $[\boldsymbol{r}_j]_{n-k}$ received from this corrupted party minus the same share that this corrupted party should hold.

     $\mathcal{S}$ reconstructs $\boldsymbol{\delta}_j$ and views it as an additive attack towards P_j's shares.
3. In Step 4, $\mathcal{S}$ follows the same strategy as that in Step 3 but only for the first $n-2k+1$ parties. For each honest party P_j, $\mathcal{S}$ extracts the additive attack towards P_j's shares, denoted by $\boldsymbol{\delta}'_j$.
4. In Step 7, for all $j \in \{n-2k+2, \ldots, n\}$, $\mathcal{S}$ computes the shares of $[\boldsymbol{r}_{n+j}]_{n-1}$ that corrupted parties should hold. Then, for each $P_j \in \{P_{n-2k+2}, \ldots, P_n\}$,
   - If P_j is corrupted, $\mathcal{S}$ samples a random vector in $\mathbb{F}^k$ as $\boldsymbol{r}_{n+j}$. Based on $\boldsymbol{r}_{n+j}$ and the shares of $[\boldsymbol{r}_{n+j}]_{n-1}$ that corrupted parties should hold, $\mathcal{S}$ randomly samples the shares of honest parties. Finally, $\mathcal{S}$ sends the shares of $\boldsymbol{r}_{n+j}$ held by honest parties to P_j.
   - If P_j is honest, $\mathcal{S}$ receives from corrupted parties their shares of $[\boldsymbol{r}_{n+j}]_{n-1}$. Then $\mathcal{S}$ sets a degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\delta}'_j]_{n-1}$ as follows:
     (a) The share of each honest party is set to be 0.
     (b) The share of each corrupted party is set to be the share of $[\boldsymbol{r}_{n+j}]_{n-1}$ received from this corrupted party minus the same share that this corrupted party should hold.

     $\mathcal{S}$ reconstructs $\boldsymbol{\delta}'_j$ and views it as an additive attack towards P_j's shares.

   Now $\mathcal{S}$ transforms the additive attacks towards the shares of honest parties to an additive attack towards the secret of the second sharing in each Π_i. For all $i \in \{1, 2, \ldots, k\}$,
   - $\mathcal{S}$ computes a degree-$(2k-2)$ packed Shamir sharing $[\boldsymbol{d}_1^{(i)} \| \texttt{pos}_i]_{2k-2}$ which is determined by using $(\delta_{j,i})_{j \in \mathcal{H}}$ as the shares of honest parties. $\boldsymbol{d}_1^{(i)}$ is viewed as an additive attack towards the secret of the first sharing in Π_i.
   - $\mathcal{S}$ computes a degree-$(2k-2)$ packed Shamir sharing $[\boldsymbol{d}_2^{(i)} \| \texttt{pos}'_i]_{2k-2}$ which is determined by using $(\delta'_{j,i})_{j \in \mathcal{H}}$ as the shares of honest parties. $\boldsymbol{d}_2^{(i)}$ is viewed as an additive attack towards the secret of the second sharing in Π_i.

   $\mathcal{S}$ computes $\boldsymbol{d}^{(i)} = \boldsymbol{d}_2^{(i)} - f_i(\boldsymbol{d}_1^{(i)})$.
5. Recall that $\mathcal{S}$ has computed $\boldsymbol{r}_j$ and $\boldsymbol{r}_{n+j}$ for all corrupted party P_j. Let $\texttt{sh}_j([\boldsymbol{d}_1^{(i)} \| \texttt{pos}_i]_{2k-2})$ denote the j-th share of $[\boldsymbol{d}_1^{(i)} \| \texttt{pos}_i]_{2k-2}$, and $\texttt{sh}_j([\boldsymbol{d}_2^{(i)} \| \texttt{pos}'_i]_{2k-2})$ denote the j-th share of $[\boldsymbol{d}_2^{(i)} \| \texttt{pos}'_i]_{2k-2}$. For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{S}$ sends $\{r_{j,i} + \texttt{sh}_j([\boldsymbol{d}_1^{(i)} \| \texttt{pos}_i]_{2k-2}), r_{n+j,i} + \texttt{sh}_j([\boldsymbol{d}_2^{(i)} \| \texttt{pos}'_i]_{2k-2})\}_{j \in \mathcal{C}orr}$ and $\boldsymbol{d}^{(i)}$ to $\mathcal{F}_{\text{rand-sharing-mal}}(\Pi_i)$.

This completes the description of the simulator. Now, we show that the simulator we constructed above perfectly simulate the behaviors of honest parties.

We first show that the messages that honest parties send to corrupted parties have the same distribution in both worlds. In Step 3, for each corrupted party P_j, honest parties need to send their shares of $[\boldsymbol{r}_j]_{n-k}$ to P_j. Recall that $[\boldsymbol{r}_j]_{n-k}$ is a random degree-$(n-k)$ packed Shamir sharing generated by $\mathcal{F}_{\text{rand}}$. In the ideal world, $\mathcal{S}$ honestly follows $\mathcal{F}_{\text{rand}}$ to compute the shares of honest parties. Therefore, the shares of $[\boldsymbol{r}_j]_{n-k}$ held by honest parties have the same distribution in both worlds. Similarly, in Step 4, for each corrupted party P_j of the first $n-2k+1$ parties, the shares of $[\boldsymbol{r}_{n+j}]_{n-k}$ held by honest parties have the same distribution in both worlds.

In Step 7, for each corrupted party $P_j \in\in \{P_{n-2k+2}, \ldots, P_n\}$, honest parties need to send their shares of $[\boldsymbol{r}_{n+j}]_{n-1}$ to P_j. Recall that $\boldsymbol{r}_{n+j}$ are used as the j-th shares of the degree-$(n-k)$ packed Shamir sharings in $\Pi_1, \ldots, \Pi_k$. Since the degree-$(n-k)$ packed Shamir secret sharing scheme has threshold t, the shares of corrupted parties are uniformly random. Therefore $\boldsymbol{r}_{n+j}$ is uniformly distributed in $\mathbb{F}^k$. Since all parties use a random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{o}_{j-n+2k-1}]_{n-1}$ as a random mask, $[\boldsymbol{r}_{n+j}]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{r}_{n+j}$ given the secret $\boldsymbol{r}_{n+j}$ and the shares of corrupted parties. In the ideal world, $\mathcal{S}$ randomly samples the secret $\boldsymbol{r}_{n+j}$ and randomly samples the shares of honest parties in $[\boldsymbol{r}_{n+j}]_{n-1}$ based on the secret $\boldsymbol{r}_{n+j}$ and the shares of corrupted parties. Thus, the shares of $[\boldsymbol{r}_{n+j}]_{n-1}$ held by honest parties have the same distribution in both worlds.

Now it is sufficient to show that the shares of the output sharings held by honest parties have the same distribution in both worlds. We note that the only thing that corrupted parties can do is to send incorrect shares to honest parties.

- In the real world, in Step 3 for each honest party P_j, P_j uses the shares he received as a degree-$(n-1)$ packed Shamir sharing and reconstructs $\tilde{\boldsymbol{r}}_j$ (Here we use the tilde notation to distinguish from the correct secret $\boldsymbol{r}_j$ that P_j should obtain.) Compared with the shares that P_j should received, the difference is the degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{\delta}_j]_{n-1}$ we constructed above. Thus, $\boldsymbol{\delta}_j = \tilde{\boldsymbol{r}}_j - \boldsymbol{r}_j$. In the ideal world, $\mathcal{S}$ can compute $[\boldsymbol{\delta}_j]_{n-1}$ as described above. Similarly, in Step 4 and Step 7, $\mathcal{S}$ can extract $\boldsymbol{\delta}'_j = \tilde{\boldsymbol{r}}_{n+j} - \boldsymbol{r}_{n+j}$.
- Now for all $i \in \{1, 2, \ldots, k\}$ and for each honest party P_j, the j-th share of the first sharing in Π_i is deviated by $\delta_{j,i}$ due to the malicious behaviors of corrupted parties. Let $[\boldsymbol{\tau}_i \| \mathtt{pos}_i]_{n-1}$ denote the first sharing that all parties should obtain. Then the shares that honest parties actually obtain are equal to the shares of $[\boldsymbol{\tau}_i \| \mathtt{pos}_i]_{n-1} + [\boldsymbol{d}_1^{(i)} \| \mathtt{pos}_i]_{2k-2}$, where recall that $[\boldsymbol{d}_1^{(i)} \| \mathtt{pos}_i]_{2k-2}$ is the degree-$(2k-2)$ packed Shamir sharing determined by using $(\delta_{j,i})_{j \in \mathcal{H}}$ as the shares of honest parties. Note that when $k \leq (n+2)/3$, we have $2k-2 \leq n-1$. Therefore $[\boldsymbol{\tau}_i \| \mathtt{pos}_i]_{n-1} + [\boldsymbol{d}_1^{(i)} \| \mathtt{pos}_i]_{2k-2} = [\boldsymbol{\tau}_i + \boldsymbol{d}_1^{(i)} \| \mathtt{pos}_i]_{n-1}$. Since $[\boldsymbol{\tau}_i \| \mathtt{pos}_i]_{n-1}$ is a random degree-$(n-1)$ packed Shamir sharing given the shares of corrupted parties, $[\boldsymbol{\tau}_i + \boldsymbol{d}_1^{(i)} \| \mathtt{pos}_i]_{n-1}$ is also a random degree-$(n-1)$ packed Shamir sharing given the shares of corrupted parties. Thus, by sending the shares of $[\boldsymbol{\tau}_i + \boldsymbol{d}_1^{(i)} \| \mathtt{pos}_i]_{n-1}$ of corrupted parties to $\mathcal{F}_{\text{rand-sharing-mal}}$, $\mathcal{F}_{\text{rand-sharing-mal}}$ generates the shares of $[\boldsymbol{\tau}_i + \boldsymbol{d}_1^{(i)} \| \mathtt{pos}_i]_{n-1}$ of honest parties with the same distribution as that in the real world.
  Similarly, for the second sharing $[f_i(\boldsymbol{\tau}_i) \| \mathtt{pos}'_i]_{n-k}$ and for each honest party P_j, the j-th share is deviated by $\delta'_{j,i}$ due to the malicious behaviors of corrupted parties. Then the

shares that honest parties actually obtain are equal to the shares of $[f_i(\boldsymbol{\tau}_i)\|\mathtt{pos}'_i]_{n-k} + [\boldsymbol{d}_2^{(i)}\|\mathtt{pos}'_i]_{2k-2}$, where recall that $[\boldsymbol{d}_2^{(i)}\|\mathtt{pos}'_i]_{2k-2}$ is the degree-$(2k-2)$ packed Shamir sharing determined by using $(\delta'_{j,i})_{j\in\mathcal{H}}$ as the shares of honest parties. Note that when $k \le (n+2)/3$, we have $2k-2 \le n-k$. Therefore $[f_i(\boldsymbol{\tau}_i)\|\mathtt{pos}'_i]_{n-k} + [\boldsymbol{d}_2^{(i)}\|\mathtt{pos}'_i]_{2k-2} = [f_i(\boldsymbol{\tau}_i)+\boldsymbol{d}_2^{(i)}\|\mathtt{pos}'_i]_{n-k}$. Note that a degree-$(n-k)$ packed Shamir sharing is determined by the secret and the shares of corrupted parties. Since we have provided the shares of $[\boldsymbol{\tau}_i + \boldsymbol{d}_1^{(i)}\|\mathtt{pos}_i]_{n-1}$ of corrupted parties to $\mathcal{F}_{\text{rand-sharing-mal}}$, the secret generated by $\mathcal{F}_{\text{rand-sharing-mal}}$ would correspond to $\boldsymbol{\tau}_i + \boldsymbol{d}_1^{(i)}$. Then the additive errors to the secret becomes $(f_i(\boldsymbol{\tau}_i) + \boldsymbol{d}_2^{(i)}) - f_i(\boldsymbol{\tau}_i + \boldsymbol{d}_1^{(i)}) = \boldsymbol{d}_2^{(i)} - f_i(\boldsymbol{d}_1^{(i)}) = \boldsymbol{d}^{(i)}$. Thus, by sending the shares of $[f_i(\boldsymbol{\tau}_i) + \boldsymbol{d}_2^{(i)}\|\mathtt{pos}'_i]_{n-k}$ of corrupted parties and the additive errors $\boldsymbol{d}^{(i)}$, $\mathcal{F}_{\text{rand-sharing-mal}}$ generates the shares of $[f_i(\boldsymbol{\tau}_i) + \boldsymbol{d}_2^{(i)}\|\mathtt{pos}'_i]_{n-k}$ of honest parties with the same distribution as that in the real world.

We conclude that when $k \le (n+2)/3$, Protocol RAND-SHARING-MAL securely computes $\mathcal{F}_{\text{rand-sharing-mal}}$ in the $\{\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{randZero}}\}$-hybrid model against a fully malicious adversary who controls $t = n - 2k + 1$ parties. □

14.1.2 Performing Sharing Transformation

Now we present the concrete protocol for performing the sharing transformation that we are interested in. We modify the protocol TRAN by requiring that P_1 distributes a degree-$(2k-2)$ packed Shamir sharing of the reconstruction result. In this way, the whole sharing is determined by the shares of honest parties. Note that when $k \le (n+2)/3$, we have $2k-2 \le n-k$. Thus, all parties can still obtain a degree-$(n-k)$ Shamir sharing at the end.

The description of the protocol TRAN-MAL appears in Protocol 14.3.

Figure 14.3: Protocol TRAN-MAL

1. All parties agree on the tuple $(\mathtt{pos}, \mathtt{pos}', f)$ where $\mathtt{pos}, \mathtt{pos}'$ are two vectors of field elements in $\mathbb{F}^k$ and $f : \mathbb{F}^k \to \mathbb{F}^k$ is a linear map. All parties start with a degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{x}\|\mathtt{pos}]_{n-1}$. The goal is to compute $[f(\boldsymbol{x})\|\mathtt{pos}']_{n-k}$.
2. Let $\Pi = \Pi(\mathtt{pos}, \mathtt{pos}', f)$ defined in Chapter 14.1.1. All parties invoke $\mathcal{F}_{\text{rand-sharing-mal}}(\Pi)$ to prepare a random Π-sharing $([\boldsymbol{r}\|\mathtt{pos}]_{n-1}, [f(\boldsymbol{r})\|\mathtt{pos}']_{n-k})$.
3. All parties locally compute $[\boldsymbol{x}+\boldsymbol{r}\|\mathtt{pos}]_{n-1} = [\boldsymbol{x}\|\mathtt{pos}]_{n-1} + [\boldsymbol{r}\|\mathtt{pos}]_{n-1}$ and send their shares to the first party P_1.
4. P_1 reconstructs the secret $\boldsymbol{x}+\boldsymbol{r}$. Then P_1 computes $f(\boldsymbol{x}+\boldsymbol{r})$ and generates a degree-$(2k-2)$ packed Shamir sharing $[f(\boldsymbol{x}+\boldsymbol{r})\|\mathtt{pos}']_{2k-2}$. Finally, P_1 distributes the shares of $[f(\boldsymbol{x}+\boldsymbol{r})\|\mathtt{pos}']_{2k-2}$ to all parties.
5. All parties locally compute $[f(\boldsymbol{x})\|\mathtt{pos}']_{n-k} = [f(\boldsymbol{x}+\boldsymbol{r})\|\mathtt{pos}']_{2k-2} - [f(\boldsymbol{r})\|\mathtt{pos}']_{n-k}$.

14.2 Circuit-Independent Preprocessing Phase

In the circuit independent preprocessing phase, all parties prepare the following correlated random sharings:

- All parties prepare a random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{\gamma}]_{n-k}$, where $\boldsymbol{\gamma} = (\gamma, \gamma, \ldots, \gamma) \in \mathbb{F}^k$ and γ is a random field element, which is served as the MAC key.
- For every group of k input gates and output gates:
  1. All parties prepare an authenticated random degree-$(n-k)$ packed Shamir sharings $[\![\boldsymbol{r}]\!]_{n-k} = ([\boldsymbol{r}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k})$. In addition, all parties prepare another random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{\Delta}]_{n-k}$ and compute the sharing $[\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$. Note that $([\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k})$ can be seen as an authentication of the sharing $[\boldsymbol{r}]_{n-k}$ under the MAC key $[\boldsymbol{\Delta}]_{n-k}$. We will use $([\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k})$ to allow a client to verify the correctness of the secret $\boldsymbol{r}$ in the input protocol and the output protocol.
  2. For every group of k output gates, all parties also prepare a random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0}$, denoted by $[\boldsymbol{o}]_{n-1}$. We will use $[\boldsymbol{o}]_{n-1}$ to protect the shares of honest parties.
- For every group of k multiplication gates:
  1. All parties prepare an authenticated packed Beaver triple $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$ where $\boldsymbol{a}, \boldsymbol{b}$ are random vectors in $\mathbb{F}^k$ and $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$.
  2. All parties prepare two random degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$, denoted by $[\boldsymbol{o}^{(1)}]_{n-1}$, $[\boldsymbol{o}^{(2)}]_{n-1}$. In the multiplication protocol, these sharings are used as random masks to protect the shares of honest parties.
- All parties also prepare the following sharings for the verification of the computation: All parties prepare two random degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$, denoted by $[\boldsymbol{o}^{(1)}]_{n-1}$ and $[\boldsymbol{o}^{(2)}]_{n-1}$. These two sharings are used to protect the shares of honest parties when checking the correctness of multiplications.

The functionality $\mathcal{F}_{\text{prep-mal}}$ for the circuit independent preprocessing phase appears in Functionality 14.4. The size of the preprocessing data among all parties in $\mathcal{F}_{\text{prep-mal}}$ is summarized as follows:

- The sharing of the MAC key $[\boldsymbol{\gamma}]_{n-k}$: n elements.
- Per input gate: $4n/k$ elements.
- Per output gate: $5n/k$ elements.
- Per multiplication gate: $8n/k$ elements.
- For the verification of the computation: $2 \cdot n$ elements.

Figure 14.4: Functionality $\mathcal{F}_{\text{prep-mal}}$

$\mathcal{F}_{\text{prep-mal}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$. $\mathcal{F}_{\text{prep-mal}}$ samples a random field element $\gamma \in \mathbb{F}$ and sets $\boldsymbol{\gamma} = (\gamma, \gamma, \ldots, \gamma) \in \mathbb{F}^k$. Let $d \in \{n-k, n-1\}$. We define the following two procedures.

- RANDSHARING($\boldsymbol{r}$, d): $\mathcal{F}_{\text{prep-mal}}$ receives from the adversary a set of shares $\{r_j\}_{j \in \mathcal{C}orr}$. Then $\mathcal{F}_{\text{prep-mal}}$ samples a random degree-d packed Shamir sharing $[\boldsymbol{r}]_d$ such that for all $P_j \in \mathcal{C}orr$, the j-th share of $[\boldsymbol{r}]_d$ is r_j. Finally, $\mathcal{F}_{\text{prep-mal}}$ distributes the shares of $[\boldsymbol{r}]_d$ to honest parties.
- AUTHSHARING($\boldsymbol{r}$): $\mathcal{F}_{\text{prep-mal}}$ receives from the adversary a set of shares $\{(r_j, u_j)\}_{j \in \mathcal{C}orr}$. Then $\mathcal{F}_{\text{prep-mal}}$ computes two degree-$(n-k)$ packed Shamir sharings $([\boldsymbol{r}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k})$ such that for all $P_j \in \mathcal{C}orr$, the j-th shares of $([\boldsymbol{r}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k})$ are r_j, u_j respectively. Finally, $\mathcal{F}_{\text{prep-mal}}$ distributes the shares of $[\![\boldsymbol{r}]\!]_{n-k} = ([\boldsymbol{r}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k})$ to honest parties.

The ideal functionality $\mathcal{F}_{\text{prep-mal}}$ runs the following steps.

1. $\mathcal{F}_{\text{prep-mal}}$ invokes RANDSHARING($\boldsymbol{\gamma}$, $n-k$) to prepare $[\boldsymbol{\gamma}]_{n-k}$.
2. For every group of k input gates and output gates:
   (a) $\mathcal{F}_{\text{prep-mal}}$ samples a random vector $\boldsymbol{r} \in \mathbb{F}^k$ and invokes AUTHSHARING($\boldsymbol{r}$) to prepare $[\![\boldsymbol{r}]\!]_{n-k}$.
   (b) $\mathcal{F}_{\text{prep-mal}}$ samples a random vector $\boldsymbol{\Delta} \in \mathbb{F}^k$ and invokes RANDSHARING($\boldsymbol{\Delta}$, $n-k$) and RANDSHARING($\boldsymbol{\Delta} * \boldsymbol{r}$, $n-k$) to prepare $([\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k})$.
   (c) For every group of k output gates, $\mathcal{F}_{\text{prep-mal}}$ invokes RANDSHARING($\mathbf{0}$, $n-1$) to prepare $[\boldsymbol{o}]_{n-1}$, where $\boldsymbol{o} = \mathbf{0}$.
3. For every group of k multiplication gates:
   (a) $\mathcal{F}_{\text{prep-mal}}$ samples two random vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^k$ and computes $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Then, $\mathcal{F}_{\text{prep-mal}}$ invokes AUTHSHARING($\boldsymbol{a}$), AUTHSHARING($\boldsymbol{b}$), and AUTHSHARING($\boldsymbol{c}$) to prepare $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$.
   (b) $\mathcal{F}_{\text{prep-mal}}$ invokes two times of RANDSHARING($\mathbf{0}$, $n-1$) to prepare $[\boldsymbol{o}^{(1)}]_{n-1}$, $[\boldsymbol{o}^{(2)}]_{n-1}$, where $\boldsymbol{o}^{(1)} = \boldsymbol{o}^{(2)} = \mathbf{0}$.
4. All parties prepare the following random sharings for the verification of the computation: All parties invoke two times of RANDSHARING($\mathbf{0}$, $n-1$) to prepare $[\boldsymbol{o}^{(1)}]_{n-1}, [\boldsymbol{o}^{(2)}]_{n-1}$, where $\boldsymbol{o}^{(1)} = \boldsymbol{o}^{(2)} = \mathbf{0}$.

14.3 Online Computation Phase

14.3.1 Input Layer

For a group of k input gates belonging to the same client, let $\boldsymbol{x}$ be the values associated with these k input gates. Suppose pos are the positions associated with these k gates. The goal is to compute an authenticated sharing $[\![\boldsymbol{x} \| \mathtt{pos}]\!]_{n-k}$. Recall that for every group of k input gates,

all parties have prepared the random sharings $[\![\boldsymbol{r}]\!]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ in $\mathcal{F}_{\text{prep-mal}}$, where $[\![\boldsymbol{r}]\!]_{n-k} = ([\boldsymbol{r}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k})$. At a high-level, all parties first send the random sharing $[\boldsymbol{r}]_{n-k}$ to the client, and make use of the random sharings $[\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ to allow the client to verify the correctness of the secret $\boldsymbol{r}$. Then, the client samples a random degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{x} + \boldsymbol{r}]_{k-1}$ and distributes the shares to all parties. In this way, all parties can compute

$$\begin{aligned} [\boldsymbol{x}]_{n-k} &= [\boldsymbol{x} + \boldsymbol{r}]_{k-1} - [\boldsymbol{r}]_{n-k} \\ [\boldsymbol{\gamma} * \boldsymbol{x}]_{n-1} &= [\boldsymbol{x} + \boldsymbol{r}]_{k-1} \cdot [\boldsymbol{\gamma}]_{n-k} - [\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k} \end{aligned}$$

which is $[\![\boldsymbol{x}]\!]_{n-1} = ([\boldsymbol{x}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{x}]_{n-1})$. Finally, all parties apply two times of TRAN-MAL to transform $[\![\boldsymbol{x}]\!]_{n-1}$ to $[\![\boldsymbol{x} \| \mathtt{pos}]\!]_{n-k}$. The description of INPUT-MAL appears in Protocol 14.5.

Figure 14.5: Protocol INPUT-MAL

1. Suppose $\boldsymbol{x} \in \mathbb{F}^k$ is the input associated with the input gates which belongs to Client. Also suppose pos are the positions associated with these gates. Let $[\![\boldsymbol{r}]\!]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ be the random sharings prepared for these input gates in $\mathcal{F}_{\text{prep-mal}}$. Recall that $[\![\boldsymbol{r}]\!]_{n-k} = ([\boldsymbol{r}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k})$.
2. All parties send their shares of $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ to Client.
3. Client checks whether the shares of $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ form valid degree-$(n-k)$ packed Shamir sharings. If not, Client aborts. Otherwise, Client reconstructs the secret $\boldsymbol{r}, \boldsymbol{\Delta}, \boldsymbol{\Delta} * \boldsymbol{r}$ and checks the MAC of $\boldsymbol{r}$, i.e., whether the third secret is equal to the coordinate-wise multiplication of the first two secrets. If not, Client aborts.
4. If both checks pass, Client generates a random degree-$(k-1)$ packed Shamir sharing of $\boldsymbol{x} + \boldsymbol{r}$, denoted by $[\boldsymbol{x} + \boldsymbol{r}]_{k-1}$, and distributes the shares to all parties.
5. All parties locally compute

$$\begin{aligned} [\boldsymbol{x}]_{n-k} &= [\boldsymbol{x} + \boldsymbol{r}]_{k-1} - [\boldsymbol{r}]_{n-k} \\ [\boldsymbol{\gamma} * \boldsymbol{x}]_{n-1} &= [\boldsymbol{x} + \boldsymbol{r}]_{k-1} \cdot [\boldsymbol{\gamma}]_{n-k} - [\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k} \end{aligned}$$

   and set $[\![\boldsymbol{x}]\!]_{n-1} = ([\boldsymbol{x}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{x}]_{n-1})$.
6. Recall that $\boldsymbol{\beta}$ are the default positions for the packed Shamir secret sharing scheme. All parties invoke TRAN-MAL on $[\boldsymbol{x}]_{n-k}$ with $(\boldsymbol{\beta}, \mathtt{pos}, I)$, where I is the identity map, and obtain $[\boldsymbol{x} \| \mathtt{pos}]_{n-k}$. All parties invoke TRAN-MAL on $[\boldsymbol{\gamma} * \boldsymbol{x}]_{n-1}$ with $(\boldsymbol{\beta}, \mathtt{pos}, I)$ and obtain $[\boldsymbol{\gamma} * \boldsymbol{x} \| \mathtt{pos}]_{n-k}$.
7. All parties take $[\![\boldsymbol{x} \| \mathtt{pos}]\!]_{n-k} = ([\boldsymbol{x} \| \mathtt{pos}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{x} \| \mathtt{pos}]_{n-k})$ as output.

14.3.2 Network Routing

We follow the same approach as that in our semi-honest protocol to realize network routing. To obtain an authenticated degree-$(n-k)$ packed Shamir sharing $[\![\boldsymbol{x}]\!]_{n-k}$, we will apply the network

routing protocol to compute each of $([\boldsymbol{x}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{x}]_{n-k})$. The description of NETWORK-MAL appears in Protocol 14.6.

Figure 14.6: Protocol NETWORK-MAL

1. Suppose all parties want to prepare an authenticated degree-$(n-k)$ packed Shamir sharing of $\boldsymbol{x}$ stored at the default positions $\boldsymbol{\beta}$.
2. Let $x'_1, x'_2, \ldots, x'_{\ell_1}$ be the different wire values in $\boldsymbol{x}$ from previous layers. Let $c_1, c_2, \ldots, c_{\ell_2}$ be the constant values in $\boldsymbol{x}$. Let $\boldsymbol{x}' = (x'_1, \ldots, x'_{\ell_1}, c_1, \ldots, c_{\ell_2}, 0, \ldots, 0) \in \mathbb{F}^k$.
3. For all $1 \leq i \leq \ell_1$, let $[\![\boldsymbol{x}^{(i)} \| \mathtt{pos}^{(i)}]\!]_{n-k}$ be the degree-$(n-k)$ packed Shamir sharing from some previous layer that contains the secret x'_i stored at position p_i. Let $p_{\ell_1+1}, \ldots, p_k$ be the first $k - \ell_1$ distinct positions that are different from $p_1, \ldots, p_{\ell_1}$ and $\alpha_1, \ldots, \alpha_n$. (In particular, $p_i \in \{\beta_1, \ldots, \beta_k\}$ for all $i \in \{\ell_1 + 1, \ldots, k\}$.) Let $\mathtt{pos} = (p_1, \ldots, p_k)$.
4. Let $\boldsymbol{e}_i$ be the i-th unit vector in $\mathbb{F}^k$ (i.e., only the i-th term is 1 and all other terms are 0). All parties locally compute a degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{e}_i \| \mathtt{pos}]_{k-1}$.
5. All parties locally compute

$$
\begin{aligned}
[\boldsymbol{x}' \| \mathtt{pos}]_{n-1} &= \sum_{i=1}^{\ell_1} [\boldsymbol{e}_i \| \mathtt{pos}]_{k-1} \cdot [\boldsymbol{x}^{(i)} \| \mathtt{pos}^{(i)}]_{n-k} + \sum_{i=1}^{\ell_2} c_i \cdot [\boldsymbol{e}_{\ell_1+i} \| \mathtt{pos}]_{k-1} \\
[\boldsymbol{\gamma} * \boldsymbol{x}' \| \mathtt{pos}]_{n-1} &= \sum_{i=1}^{\ell_1} [\boldsymbol{e}_i \| \mathtt{pos}]_{k-1} \cdot [\boldsymbol{\gamma} * \boldsymbol{x}^{(i)} \| \mathtt{pos}^{(i)}]_{n-k} \\
&\quad + \sum_{i=1}^{\ell_2} c_i \cdot [\boldsymbol{e}_{\ell_1+i} \| \mathtt{pos}]_{k-1} \cdot [\boldsymbol{\gamma}]_{n-k}
\end{aligned}
$$

6. Let $f : \mathbb{F}^k \to \mathbb{F}^k$ be a linear map such that $\boldsymbol{x} = f(\boldsymbol{x}')$. Recall that $\boldsymbol{\beta}$ are the default positions for the packed Shamir secret sharing scheme. All parties invoke TRAN-MAL on $[\boldsymbol{x}' \| \mathtt{pos}]_{n-1}$ with $(\mathtt{pos}, \boldsymbol{\beta}, f)$ and obtain $[\boldsymbol{x}]_{n-k}$. All parties invoke TRAN-MAL on $[\boldsymbol{\gamma} * \boldsymbol{x}' \| \mathtt{pos}]_{n-1}$ with $(\mathtt{pos}, \boldsymbol{\beta}, f)$ and obtain $[\boldsymbol{\gamma} * \boldsymbol{x}]_{n-k}$.
7. All parties take $[\![\boldsymbol{x}]\!]_{n-k} = ([\boldsymbol{x}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{x}]_{n-k})$ as output.

14.3.3 Evaluating Addition Gates and Multiplication Gates

Addition Gates. We follow the same approach as that in our semi-honest protocol to evaluate addition gates. To obtain an authenticated degree-$(n-k)$ packed Shamir sharing $[\![\boldsymbol{z} \| \mathtt{pos}]\!]_{n-k}$ (the output sharing), we will invoke TRAN-MAL two times to transform $[\![\boldsymbol{z}]\!]_{n-k}$ to $[\![\boldsymbol{z} \| \mathtt{pos}]\!]_{n-k}$. The description of PACKED-ADD-MAL appears in Protocol 14.7.

Figure 14.7: Protocol PACKED-ADD-MAL

1. Suppose $[\![\boldsymbol{x}]\!]_{n-k}, [\![\boldsymbol{y}]\!]_{n-k}$ are the input packed Shamir sharings of the addition gates.
2. All parties locally compute $[\![\boldsymbol{z}]\!]_{n-k} = [\![\boldsymbol{x}]\!]_{n-k} + [\![\boldsymbol{y}]\!]_{n-k}$.
3. Suppose $\mathtt{pos}$ are the positions associated with these k addition gates. Recall that $\boldsymbol{\beta}$ are the default positions. All parties invoke TRAN-MAL on $[\boldsymbol{z}]_{n-k}$ with $(\boldsymbol{\beta}, \mathtt{pos}, I)$, where I is the identity map, and obtain $[\boldsymbol{z}\|\mathtt{pos}]_{n-k}$. All parties invoke TRAN-MAL on $[\boldsymbol{\gamma} * \boldsymbol{z}]_{n-1}$ with $(\boldsymbol{\beta}, \mathtt{pos}, I)$ and obtain $[\boldsymbol{\gamma} * \boldsymbol{z}\|\mathtt{pos}]_{n-k}$.
4. All parties take $[\![\boldsymbol{z}\|\mathtt{pos}]\!]_{n-k} = ([\boldsymbol{z}\|\mathtt{pos}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{z}\|\mathtt{pos}]_{n-k})$ as output.

Multiplication Gates. For a group of k multiplication gates, let $[\![\boldsymbol{x}]\!]_{n-k}, [\![\boldsymbol{y}]\!]_{n-k}$ be the input sharings. Recall that all parties have prepared the following random sharings in $\mathcal{F}_{\text{prep-mal}}$:

- An authenticated packed Beaver triple: $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$.
- Three degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$: $[\boldsymbol{o}^{(1)}]_{n-1}, [\boldsymbol{o}^{(2)}]_{n-1}$.

The goal is to compute an authenticated output sharing $[\![\boldsymbol{z}\|\mathtt{pos}]\!]_{n-k} = [\![\boldsymbol{x} * \boldsymbol{y}\|\mathtt{pos}]\!]_{n-k}$, where $\mathtt{pos}$ are the positions associated with these k gates. We follow a similar approach as that in our semi-honest protocol. The first difference is that, all parties will use $[\boldsymbol{o}^{(1)}]_{n-1}, [\boldsymbol{o}^{(2)}]_{n-1}$ to refresh their shares when letting P_1 reconstructs $\boldsymbol{x} + \boldsymbol{a}$ and $\boldsymbol{y} + \boldsymbol{b}$. The second difference is that P_1 not only needs to distribute degree-$(k-1)$ packed Shamir sharings of $\boldsymbol{x} + \boldsymbol{a}, \boldsymbol{y} + \boldsymbol{b}$, but also a degree-$(k-1)$ packed Shamir sharing of $(\boldsymbol{x} + \boldsymbol{a}) \cdot (\boldsymbol{y} + \boldsymbol{b})$. In this way, all parties can locally compute an authenticated degree-$(n-1)$ packed Shamir sharing of $\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{y}$. The correctness of the sharings distributed by P_1 will be checked later. Finally, we will invoke TRAN-MAL two times to transform $[\![\boldsymbol{z}]\!]_{n-1}$ to $[\![\boldsymbol{z}\|\mathtt{pos}]\!]_{n-k}$.

The description of PACKED-MULT-MAL appears in Protocol 14.8.

14.3.4 Output Layer

For a group of k output gates belonging to the same client, let $\boldsymbol{x} \in \mathbb{F}^k$ be the values associated with these k output gates. All parties hold $[\![\boldsymbol{x}]\!]_{n-k}$. Let $[\![\boldsymbol{r}]\!]_{n-k}, ([\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}), [\boldsymbol{o}]_{n-1}$ be the random sharings prepared for these k output gates in $\mathcal{F}_{\text{prep-mal}}$. To reconstruct the secret $\boldsymbol{x}$ to the client:

1. All parties send to P_1 their shares of $[\boldsymbol{x} + \boldsymbol{r}]_{n-1} := [\boldsymbol{o}]_{n-1} + [\boldsymbol{x}]_{n-k} + [\boldsymbol{r}]_{n-k}$.
2. P_1 reconstructs $\boldsymbol{x} + \boldsymbol{r}$ and distributes a degree-$(2k-2)$ packed Shamir sharing $[\boldsymbol{x} + \boldsymbol{r}]_{2k-2}$.

Before reconstructing the secret to the client, all parties verify the correctness of the computation.

Verification of the Computation. To check the correctness of the computation, it is sufficient to verify the following points:

1. For Input Phase:
    - Each degree-$(k-1)$ packed Shamir sharing distributed by a client is a valid degree-$(k-1)$ packed Shamir sharing.

Figure 14.8: Protocol PACKED-MULT-MAL

1. Suppose $[\![\boldsymbol{x}]\!]_{n-k}, [\![\boldsymbol{y}]\!]_{n-k}$ are the input packed Shamir sharings of the multiplication gates. All parties will use the following random sahrings prepared in the preprocessing phase
   - An authenticated packed Beaver triple: $([\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}, [\![\boldsymbol{c}]\!]_{n-k})$.
   - Three degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$: $[\boldsymbol{o}^{(1)}]_{n-1}, [\boldsymbol{o}^{(2)}]_{n-1}$.
2. All parties locally compute $[\boldsymbol{x}+\boldsymbol{a}]_{n-1} = [\boldsymbol{o}^{(1)}]_{n-1} + [\boldsymbol{x}]_{n-k} + [\boldsymbol{a}]_{n-k}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{n-1} = [\boldsymbol{o}^{(2)}]_{n-1} + [\boldsymbol{y}]_{n-k} + [\boldsymbol{b}]_{n-k}$.
3. The first party P_1 collects the whole sharings $[\boldsymbol{x}+\boldsymbol{a}]_{n-1}, [\boldsymbol{y}+\boldsymbol{b}]_{n-1}$ and reconstructs the secrets $\boldsymbol{x}+\boldsymbol{a}, \boldsymbol{y}+\boldsymbol{b}$. Then, P_1 computes $(\boldsymbol{x}+\boldsymbol{a})\cdot(\boldsymbol{y}+\boldsymbol{b})$ and the sharings $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{k-1}, [(\boldsymbol{x}+\boldsymbol{a})\cdot(\boldsymbol{y}+\boldsymbol{b})]_{k-1}$. Finally, P_1 distributes the shares to other parties.
4. All parties locally compute

$$\begin{aligned}
[\boldsymbol{z}]_{n-1} &= [(\boldsymbol{x}+\boldsymbol{a})\cdot(\boldsymbol{y}+\boldsymbol{b})]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1}\cdot[\boldsymbol{b}]_{n-k} \\
&\quad -[\boldsymbol{y}+\boldsymbol{b}]_{k-1}\cdot[\boldsymbol{a}]_{n-k} + [\boldsymbol{c}]_{n-k} \\
[\boldsymbol{\gamma}*\boldsymbol{z}]_{n-1} &= [\boldsymbol{\gamma}]_{n-k}\cdot[(\boldsymbol{x}+\boldsymbol{a})\cdot(\boldsymbol{y}+\boldsymbol{b})]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1}\cdot[\boldsymbol{\gamma}*\boldsymbol{b}]_{n-k} \\
&\quad -[\boldsymbol{y}+\boldsymbol{b}]_{k-1}\cdot[\boldsymbol{\gamma}*\boldsymbol{a}]_{n-k} + [\boldsymbol{\gamma}*\boldsymbol{c}]_{n-k}
\end{aligned}$$

   and set $[\![\boldsymbol{z}]\!]_{n-1} = ([\boldsymbol{z}]_{n-1}, [\boldsymbol{\gamma}*\boldsymbol{z}]_{n-1})$.
5. Suppose $\mathtt{pos}$ are the positions associated with these k multiplication gates. Recall that $\boldsymbol{\beta}$ are the default positions. All parties invoke TRAN-MAL on $[\boldsymbol{z}]_{n-1}$ with $(\boldsymbol{\beta}, \mathtt{pos}, I)$, where I is the identity map, and obtain $[\boldsymbol{z}\|\mathtt{pos}]_{n-k}$. All parties invoke TRAN-MAL on $[\boldsymbol{\gamma}*\boldsymbol{z}]_{n-1}$ with $(\boldsymbol{\beta}, \mathtt{pos}, I)$ and obtain $[\boldsymbol{\gamma}*\boldsymbol{z}\|\mathtt{pos}]_{n-k}$.
6. All parties take $[\![\boldsymbol{z}\|\mathtt{pos}]\!]_{n-k} = ([\boldsymbol{z}\|\mathtt{pos}]_{n-k}, [\boldsymbol{\gamma}*\boldsymbol{z}\|\mathtt{pos}]_{n-k})$ as output.

   - All parties obtain correct $[\![\boldsymbol{x}\|\mathtt{pos}]\!]_{n-k}$ from $[\![\boldsymbol{x}]\!]_{n-1}$ when using TRAN.
2. For Network Routing: All parties obtain correct $[\![\boldsymbol{x}]\!]_{n-k}$ from $[\![\boldsymbol{x}'\|\mathtt{pos}]\!]_{n-1}$ when using TRAN.
3. For Addition Gates: All parties obtain a correct output sharing $[\![\boldsymbol{z}\|\mathtt{pos}]\!]_{n-k}$ from $[\![\boldsymbol{z}]\!]_{n-k}$ when using TRAN.
4. For Multiplication Gates:
   - $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{k-1}, [(\boldsymbol{x}+\boldsymbol{a})\cdot(\boldsymbol{y}+\boldsymbol{b})]_{k-1}$ are valid degree-$(k-1)$ packed Shamir sharings. The first two secrets are identical to the secrets of $[\![\boldsymbol{x}+\boldsymbol{a}]\!]_{n-k} = [\![\boldsymbol{x}]\!]_{n-k} + [\![\boldsymbol{a}]\!]_{n-k}$ and $[\![\boldsymbol{y}+\boldsymbol{b}]\!]_{n-k} = [\![\boldsymbol{y}]\!]_{n-k} + [\![\boldsymbol{b}]\!]_{n-k}$, and the third secret is equal to the coordinate-wise multiplication between the first two secrets.
   - All parties obtain a correct output sharing $[\![\boldsymbol{z}\|\mathtt{pos}]\!]_{n-k}$ from $[\![\boldsymbol{z}]\!]_{n-1}$ when using

TRAN.

5. For Output Gates: The secret of $[\boldsymbol{x}+\boldsymbol{r}]_{2k-2}$ is identical to the secret of $[\![\boldsymbol{x}+\boldsymbol{r}]\!]_{n-k} = [\![\boldsymbol{x}]\!]_{n-k} + [\![\boldsymbol{r}]\!]_{n-k}$.

We will use an ideal functionality $\mathcal{F}_{\text{com}}$ that allows all parties to commit their values. The description of $\mathcal{F}_{\text{com}}$ appears in Functionality 14.9. It has been shown in [30] that $\mathcal{F}_{\text{com}}$ can be realized with the cost of $2n^2+n$ elements of preprocessing data and $4n^2$ elements of communication.

Figure 14.9: Functionality $\mathcal{F}_{\text{com}}$

1. On input $(\texttt{Commit}, v, i, \tau_v)$ by P_i, $\mathcal{F}_{\text{com}}$ stores (v, i, τ_v) and outputs (i, τ_v) to all parties, where τ_v represents a handle for the commitment.
2. On input $(\texttt{Open}, i, \tau_v)$ by P_i, $\mathcal{F}_{\text{com}}$ outputs (v, i, τ_v) to all parties if exists, or `abort` otherwise.

We will first verify that all degree-$(k-1)$ packed Shamir sharings are valid.

Verification of Degree-$(k-1)$ Packed Shamir Sharings. The verification is done by computing a random linear combination of all degree-$(k-1)$ packed Shamir sharings and checking the correctness of the resulting degree-$(k-1)$ packed Shamir sharing by each party. Note that we do not need to protect the secrecy of these sharings since they are generated by P_1 or a client, who can be corrupted. The description of CHECK-CONSISTENCY appears in Protocol 14.10. Recall that the cost of $\mathcal{F}_{\text{com}}$ is $2n^2+n$ elements of preprocessing data and $4n^2$ elements of communication. The total cost of CHECK-CONSISTENCY is $2n^3+n^2$ elements of preprocessing data and $4n^3+n^2$ elements of communication.

Lemma 14.2. *If there exists $i \in \{1,2,\ldots,m\}$ such that the shares of $[\boldsymbol{x}^{(i)}]_{k-1}$ of honest parties do not correspond to a valid degree-$(k-1)$ packed Shamir sharing, with probability at least $1-m/|\mathbb{F}|$, all honest parties abort in* CHECK-CONSISTENCY.

Proof. By $\mathcal{F}_{\text{com}}$, all parties obtain a uniform element λ in Step 3. Consider the polynomial

$$[\boldsymbol{F}(\lambda)]_{k-1} = [\boldsymbol{x}^{(1)}]_{k-1} + [\boldsymbol{x}^{(2)}]_{k-1} \cdot \lambda + \ldots + [\boldsymbol{x}^{(m)}]_{k-1} \cdot \lambda^{m-1}.$$

By Lagrange interpolation, for any m different points $\lambda^{(1)}, \lambda^{(2)}, \ldots, \lambda^{(m)}$, there is a one-to-one linear map from $\{[\boldsymbol{F}(\lambda^{(i)})]_{k-1}\}_{i=1}^{m}$ to $\{[\boldsymbol{x}^{(i)}]_{k-1}\}_{i=1}^{m}$. Thus, if there exists $i \in \{1,2,\ldots,m\}$ such that the shares of $[\boldsymbol{x}^{(i)}]_{k-1}$ of honest parties do not correspond to a valid degree-$(k-1)$ packed Shamir sharing, then the number of $\lambda \in \mathbb{F}$ such that the shares of $[\boldsymbol{F}(\lambda)]_{k-1}$ of honest parties correspond to a valid degree-$(k-1)$ packed Shamir sharing is bounded by $m-1$. Since λ is a uniform element in $\mathbb{F}$, the probability of sampling such a λ is bounded by $(m-1)/|\mathbb{F}| \leq m/|\mathbb{F}|$. Note that for every λ where the shares of $[\boldsymbol{F}(\lambda)]_{k-1}$ of honest parties do not correspond to a valid degree-$(k-1)$ packed Shamir sharing, all honest parties will abort since they will always receive the correct shares of honest parties. Thus, the lemma holds. □

Figure 14.10: Protocol CHECK-CONSISTENCY

1. Let m be the number of degree-$(k-1)$ packed Shamir sharings that all parties need to check. These m sharings are denoted by
$$[\boldsymbol{x}^{(1)}]_{k-1}, [\boldsymbol{x}^{(2)}]_{k-1}, \ldots, [\boldsymbol{x}^{(m)}]_{k-1}.$$
2. Each party P_i samples a random field element $\lambda_i \in \mathbb{F}$ and invokes $\mathcal{F}_{\text{com}}$ with $(\texttt{Commit}, \lambda_i, i, \tau_{\lambda_i})$.
3. Each party P_i invokes $\mathcal{F}_{\text{com}}$ with $(\texttt{Open}, i, \tau_{\lambda_i})$. All parties set $\lambda = \lambda_1 + \lambda_2 + \ldots + \lambda_n$.
4. All parties locally compute
$$[\boldsymbol{x}]_{k-1} = [\boldsymbol{x}^{(1)}]_{k-1} + [\boldsymbol{x}^{(2)}]_{k-1} \cdot \lambda + \ldots + [\boldsymbol{x}^{(m)}]_{k-1} \cdot \lambda^{m-1}.$$
5. All parties send their shares of $[\boldsymbol{x}]_{k-1}$ to all other parties. Then each party P_i checks whether the shares of $[\boldsymbol{x}]_{k-1}$ form a valid degree-$(k-1)$ packed Shamir sharing. If not, P_i aborts. Otherwise, P_i accepts.

Verification of the Secrets of Degree-$(2k-2)$ Packed Shamir Sharings. In this part, for a pair of sharings $([\boldsymbol{x}]_{2k-2}, [\![\boldsymbol{x}]\!]_{n-k})$, we want to verify that the secret of $[\boldsymbol{x}]_{2k-2}$ is identical to that authenticated in $[\![\boldsymbol{x}]\!]_{n-k}$. We assume that we do not need to protect the privacy of the secret.

This can be used to verify the secrets of $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{k-1}$ for multiplication gates (note that we can always view a degree-$(k-1)$ packed Shamir sharing as a degree-$(2k-2)$ packed Shamir sharing), and the secret of $[\boldsymbol{x}+\boldsymbol{r}]_{2k-2}$ for output gates. Also for the correctness of $[(\boldsymbol{x}+\boldsymbol{a})\cdot(\boldsymbol{y}+\boldsymbol{b})]_{k-1}$, note that $[(\boldsymbol{x}+\boldsymbol{a})\cdot(\boldsymbol{y}+\boldsymbol{b})]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1}$ should be a degree-$(2k-2)$ packed Shamir sharing of $\mathbf{0}$. We can also verify the secret of $[(\boldsymbol{x}+\boldsymbol{a})\cdot(\boldsymbol{y}+\boldsymbol{b})]_{k-1}$.

At a high-level, we simply compute a random linear combination of all pairs of random sharings and then reconstruct the resulting pair of sharings. The verification protocol will use the two degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$, $[\boldsymbol{o}^{(1)}]_{n-1}$ and $[\boldsymbol{o}^{(2)}]_{n-1}$, prepared in $\mathcal{F}_{\text{prep-mal}}$. The description of CHECK-SECRET appears in Protocol 14.11. Recall that the cost of $\mathcal{F}_{\text{com}}$ is $2n^2+n$ elements of preprocessing data and $4n^2$ elements of communication. The total cost of CHECK-SECRET is $8n^3+4n^2$ elements of preprocessing data and $16n^3+n^2$ elements of communication.

Informally, the effectiveness of CHECK-SECRET comes from the following two points:

- For all $i \in \{1,2,\ldots,m\}$, corrupted parties cannot change the secret of $[\boldsymbol{u}^{(i)}]_{2k-2}$ and the secret authenticated by $[\![\boldsymbol{u}^{(i)}]\!]_{n-k}$. The former is because the secret of a degree-$(2k-2)$ packed Shamir sharing is determined by the shares of honest parties. The latter is due to the security of the information-theoretic MAC.
- If there exists $i \in \{1,2,\ldots,m\}$ such that the secret of $[\boldsymbol{u}^{(i)}]_{2k-2}$ is different from the secret authenticated by $[\![\boldsymbol{u}^{(i)}]\!]_{n-k}$, with overwhelming probability (when the underlying field $\mathbb{F}$ is large enough), the secret of $[\boldsymbol{u}]_{2k-2}$ is different from the secret authenticated by

Figure 14.11: Protocol CHECK-SECRET

1. Let $\{([\boldsymbol{u}^{(i)}]_{2k-2}, [\![\boldsymbol{u}^{(i)}]\!]_{n-k})\}_{i=1}^{m}$ be the m pairs of sharings that all parties want to check. Let $[\boldsymbol{o}^{(1)}]_{n-1}, [\boldsymbol{o}^{(2)}]_{n-1}$ be the random degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$ prepared in $\mathcal{F}_{\text{prep-mal}}$, and $[\boldsymbol{\gamma}]_{n-k}$ be the degree-$(n-k)$ packed Shamir sharing of the authentication key prepared in $\mathcal{F}_{\text{prep-mal}}$.
2. Each party P_i samples a random field element $\lambda_i \in \mathbb{F}$ and invokes $\mathcal{F}_{\text{com}}$ with $(\texttt{Commit}, \lambda_i, i, \tau_{\lambda_i})$.
3. Each party P_i invokes $\mathcal{F}_{\text{com}}$ with $(\texttt{Open}, i, \tau_{\lambda_i})$. All parties set $\lambda = \lambda_1 + \lambda_2 + \ldots + \lambda_n$.
4. All parties locally compute

$$\begin{aligned}
[\boldsymbol{u}]_{2k-2} &= [\boldsymbol{u}^{(1)}]_{2k-2} + [\boldsymbol{u}^{(2)}]_{2k-2} \cdot \lambda + \ldots + [\boldsymbol{u}^{(m)}]_{2k-2} \cdot \lambda^{m-1}, \\
[\boldsymbol{u}]_{n-1} &= [\boldsymbol{o}^{(1)}]_{n-1} + [\boldsymbol{u}^{(1)}]_{n-k} + [\boldsymbol{u}^{(2)}]_{n-k} \cdot \lambda + \ldots + [\boldsymbol{u}^{(m)}]_{n-k} \cdot \lambda^{m-1}, \\
[\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1} &= [\boldsymbol{o}^{(2)}]_{n-1} + [\boldsymbol{\gamma} * \boldsymbol{u}^{(1)}]_{n-k} + [\boldsymbol{\gamma} * \boldsymbol{u}^{(2)}]_{n-k} \cdot \lambda + \ldots \\
&\quad + [\boldsymbol{\gamma} * \boldsymbol{u}^{(m)}]_{n-k} \cdot \lambda^{m-1}.
\end{aligned}$$

5. All parties send their shares of $[\boldsymbol{u}]_{2k-2}$ to all other parties. All parties also commit their shares of $[\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma}]_{n-k}$ using $\mathcal{F}_{\text{com}}$.
6. All parties open their shares of $[\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma}]_{n-k}$ using $\mathcal{F}_{\text{com}}$. Then each party P_i checks the following:
   (a) The shares of $[\boldsymbol{u}]_{2k-2}$ form a valid degree-$(2k-2)$ packed Shamir sharing.
   (b) The shares of $[\boldsymbol{\gamma}]_{n-k}$ form a valid degree-$(n-k)$ packed Shamir sharing and the secret $\boldsymbol{\gamma}$ is in the form of $(\gamma, \gamma, \ldots, \gamma) \in \mathbb{F}^k$.
   (c) The secrets of $[\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma}]_{n-k}$ satisfy that the second secret is equal to the coordinate-wise multiplication of the first secret and the third secret.
   (d) The secret of $[\boldsymbol{u}]_{2k-2}$ is the same as the secret of $[\boldsymbol{u}]_{n-1}$.

   If all checks pass, P_i accepts. Otherwise, P_i aborts.

$[\![\boldsymbol{u}]\!]_{n-1} = ([\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1})$.

The formal argument is deferred to the final proof of the whole protocol.

Security of TRAN. In the final proof of the whole protocol, we will show that what an adversary can do in TRAN is to insert an additive error to the secret of the resulting sharing. Since we obtain $[\![f(\boldsymbol{x})\|\texttt{pos}]\!]_{n-k}$ from $[\![\boldsymbol{x}\|\texttt{pos}]\!]_{n-1}$ by invoking TRAN on $[\boldsymbol{x}\|\texttt{pos}]_{n-1}$ and $[\boldsymbol{\gamma} * \boldsymbol{x}\|\texttt{pos}]_{n-1}$ separately, by the security of the information-theoretic MAC, any additive errors inserted by the adversary will lead to an invalid authenticated degree-$(n-k)$ packed Shamir sharing $[\![f(\boldsymbol{x})\|\texttt{pos}]\!]_{n-k}$ with overwhelming probability. Such attacks can be detected when verifying that the reconstruction of the authenticated sharings when evaluating multiplication gates and output

gates are correct, which is covered by Protocol CHECK-SECRET.

Reconstructing the Output. After all parties verify the computation, they reconstruct the function output to the clients. For a group of k output gates belonging to the same client, let $\boldsymbol{x} \in \mathbb{F}^k$ be the values associated with these k output gates. All parties hold $[\![\boldsymbol{x}]\!]_{n-k}$. Let $[\![\boldsymbol{r}]\!]_{n-k}, ([\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}), [\boldsymbol{o}]_{n-1}$ be the random sharings prepared for these k output gates in $\mathcal{F}_{\text{prep-mal}}$. Recall that all parties have obtained a degree-$(2k-2)$ packed Shamir sharing $[\boldsymbol{x}+\boldsymbol{r}]_{2k-2}$. To reconstruct the secret $\boldsymbol{x}$ to the client:

1. All parties send their shares of $[\boldsymbol{x}+\boldsymbol{r}]_{2k-2}$ to the client to allow him to reconstruct the secret $\boldsymbol{x}+\boldsymbol{r}$.
2. All parties send their shares of $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ to the client to allow him to reconstruct and verify the secret $\boldsymbol{r}$.

Finally, the client can compute $\boldsymbol{x} = (\boldsymbol{x}+\boldsymbol{r}) - \boldsymbol{r}$. The description of OUTPUT-MAL appears in Protocol 14.12.

Figure 14.12: Protocol OUTPUT-MAL

1. Suppose $\boldsymbol{x} \in \mathbb{F}^k$ is the output associated with the output gates which belongs to Client. Let $[\![\boldsymbol{r}]\!]_{n-k}, ([\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}), [\boldsymbol{o}]_{n-1}$ be the random sharings prepared for these output gates in $\mathcal{F}_{\text{prep-mal}}$. Recall that $[\![\boldsymbol{r}]\!]_{n-k} = ([\boldsymbol{r}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k})$. All parties hold $[\boldsymbol{x}+\boldsymbol{r}]_{2k-2}$ at the beginning of the protocol.
2. All parties send their shares of $[\boldsymbol{x}+\boldsymbol{r}]_{2k-2}, [\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ to Client.
3. Client checks whether the shares of $[\boldsymbol{x}+\boldsymbol{r}]_{2k-2}$ form a valid degree-$(2k-2)$ packed Shamir sharing. If not, Client aborts. Otherwise, Client reconstructs the secret $\boldsymbol{x}+\boldsymbol{r}$.
4. Client checks whether the shares of $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ form valid degree-$(n-k)$ packed Shamir sharings. If not, Client aborts. Otherwise, Client reconstructs the secret $\boldsymbol{r}, \boldsymbol{\Delta}, \boldsymbol{\Delta} * \boldsymbol{r}$ and checks the MAC of $\boldsymbol{r}$, i.e., whether the third secret is equal to the coordinate-wise multiplication of the first two secrets. If not, Client aborts.
5. If all checks pass, Client computes $\boldsymbol{x} = (\boldsymbol{x}+\boldsymbol{r}) - \boldsymbol{r}$.

14.4 Main Protocol

Now we are ready to introduce the main protocol. We restate the ideal functionality $\mathcal{F}_{\text{main-mal}}$ below. The description of PACKED-MAIN-MAL appears in Protocol 14.13.

Lemma 14.3. *For all $k \leq (n+2)/3$, Protocol* PACKED-MAIN-MAL *securely computes $\mathcal{F}_{\text{main-mal}}$ in the $\{\mathcal{F}_{\text{prep-mal}}, \mathcal{F}_{\text{rand-sharing-mal}}, \mathcal{F}_{\text{com}}\}$-hybrid model against a fully malicious adversary who controls $t = n - 2k + 1$ parties.*

Figure 7.11: Functionality $\mathcal{F}_{\text{main-mal}}$

1. $\mathcal{F}_{\text{main-mal}}$ receives from all clients their inputs.
2. $\mathcal{F}_{\text{main-mal}}$ evaluates the circuits and computes the output. $\mathcal{F}_{\text{main-mal}}$ first sends the output of corrupted clients to the adversary.
   - If the adversary replies `continue`, $\mathcal{F}_{\text{main-mal}}$ distributes the output to honest clients.
   - If the adversary replies `abort`, $\mathcal{F}_{\text{main-mal}}$ sends `abort` to honest clients.

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}orr$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties.

Throughout the simulator, $\mathcal{S}$ will keep tracking the shares that should be held by corrupted parties. We will maintain the invariant that for all degree-$(2k-2)$ packed Shamir sharings (including degree-$(k-1)$ packed Shamir sharings), $\mathcal{S}$ will learn the whole sharings and their secrets. For every authenticated degree-$(n-k)$ packed Shamir sharing $[\![\boldsymbol{x}\|\mathtt{pos}]\!]_{n-k}$, $\mathcal{S}$ will compute a pair of vectors, which are the additive errors to the secret and its MAC due to the behaviors of corrupted parties.

Simulation of PACKED-MAIN-MAL. In the preprocessing phase, $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{prep-mal}}$ and receives the shares of corrupted parties.

Simulating the Input Phase. In Step 3, for every group of k input gates of Client_i, $\mathcal{S}$ simulates INPUT-MAL. In Step 1 of INPUT-MAL, recall that $\mathcal{S}$ has learnt the shares of $[\![\boldsymbol{r}]\!]_{n-k}$, $([\boldsymbol{\Delta}]_{n-k}$, $[\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k})$ of corrupted parties when emulating $\mathcal{F}_{\text{prep-mal}}$. Starting from Step 2 of INPUT-MAL, there are two cases:

- If Client_i is honest, in Step 2, $\mathcal{S}$ receives the shares of $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ of corrupted parties.
  In Step 3, for each sharing $[\boldsymbol{p}]_{n-k} \in \{[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}\}$, $\mathcal{S}$ computes a sharing $[\delta(\boldsymbol{p})]_{n-k}$ which is defined as follows:
  1. The shares of all honest parties are set to be 0.
  2. For each corrupted party P_j, the j-th share is equal to the j-th share of $[\boldsymbol{p}]_{n-k}$ received from P_j minus the same share that P_j should hold.

  $\mathcal{S}$ checks whether $[\delta(\boldsymbol{p})]_{n-k}$ is a valid degree-$(n-k)$ packed Shamir sharing of $\mathbf{0}$. If not, $\mathcal{S}$ aborts on behalf of Client_i.
  Otherwise, in Step 4, $\mathcal{S}$ generates a random vector as $\boldsymbol{x} + \boldsymbol{r}$ and honestly follows the protocol by generating a random degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{x} + \boldsymbol{r}]_{k-1}$ and distributing the shares to corrupted parties.
  In Step 5, $\mathcal{S}$ computes the shares of $[\![\boldsymbol{x}]\!]_{n-1}$ held by corrupted parties.
- If Client_i is corrupted, in Step 2, $\mathcal{S}$ samples two random vectors $\boldsymbol{r}, \boldsymbol{\Delta}$ and computes the whole sharings $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$. Then $\mathcal{S}$ sends the shares of honest parties to

Figure 14.13: Protocol PACKED-MAIN-MAL

1. **Preprocessing Phase**. All parties invoke $\mathcal{F}_{\text{prep-mal}}$ to prepare correlated randomness for the online phase.
2. **Initialization**. Let $\alpha_1, \ldots, \alpha_n$ be distinct field elements in $\mathbb{F}$ which are used for the shares of all parties in packed Shamir secret sharing schemes. Let $\beta_1, \ldots, \beta_{|C|}$ be $|C|$ distinct field elements that are different from $\alpha_1, \ldots, \alpha_n$. Let $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_k)$ be the default positions for the packed Shamir secret sharing schemes. We associate the field element β_i with the i-th gate in C.
3. **Input Phase**. Let $\mathsf{Client}_1, \ldots, \mathsf{Client}_\mathsf{c}$ denote the clients who provide inputs. All input gates are divided into groups of size k based on the input holders. For every group of k input gates of Client_i, suppose $\boldsymbol{x}$ are the inputs, and $\mathtt{pos} = (p_1, \ldots, p_k)$ are the positions associated with these k gates. All parties and Client_i invokes INPUT-MAL to obtain $[\![\boldsymbol{x} \| \mathtt{pos}]\!]_{n-k}$.
4. **Evaluation Phase**. All parties evaluate the circuit layer by layer as follows:
   (a) For the current layer, all gates are divided into groups of size k based on their types (i.e., multiplication gates or addition gates). For each group of k gates, let $\boldsymbol{x}, \boldsymbol{y}$ be the inputs of all gates. All parties invoke NETWORK-MAL to prepare $[\![\boldsymbol{x}]\!]_{n-k}$ and $[\![\boldsymbol{y}]\!]_{n-k}$.
   (b) For each group of k gates, let $\mathtt{pos}$ be the positions associated with these k gates. For addition gates, all parties invoke PACKED-ADD-MAL on $([\![\boldsymbol{x}]\!]_{n-k}, [\![\boldsymbol{y}]\!]_{n-k})$, and obtain $[\![\boldsymbol{z} \| \mathtt{pos}]\!]_{n-k}$. For multiplication gates, all parties invoke PACKED-MULT-MAL on $([\![\boldsymbol{x}]\!]_{n-k}, [\![\boldsymbol{y}]\!]_{n-k})$ and obtain $[\![\boldsymbol{z} \| \mathtt{pos}]\!]_{n-k}$.
5. **Output Phase**. All output gates are divided into groups of size k based on the output receiver. For every group of k output gates of Client_i, suppose $\boldsymbol{x}$ are the inputs. All parties invoke the protocol NETWORK-MAL to prepare $[\![\boldsymbol{x}]\!]_{n-k}$. Then, all parties run the following steps.
   (a) Let $[\![\boldsymbol{r}]\!]_{n-k}, ([\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}), [\boldsymbol{o}]_{n-1}$ be the random sharings prepared for these k output gates in $\mathcal{F}_{\text{prep-mal}}$. All parties send to P_1 their shares of $[\boldsymbol{x}+\boldsymbol{r}]_{n-1} := [\boldsymbol{o}]_{n-1} + [\boldsymbol{x}]_{n-k} + [\boldsymbol{r}]_{n-k}$.
   (b) P_1 reconstructs $\boldsymbol{x} + \boldsymbol{r}$ and distributes a degree-$(2k-2)$ packed Shamir sharing $[\boldsymbol{x} + \boldsymbol{r}]_{2k-2}$.

   Now all parties verify the computation. (1) All parties invoke CHECK-CONSISTENCY to check the correctness of all degree-$(k-1)$ packed Shamir sharings. (2) All parties invoke CHECK-SECRET to check the correctness of the secrets of the degree-$(2k-2)$ (including degree-$(k-1)$) packed Shamir sharings generated by P_1.

   If all parties accept the verification, all parties and Client_i invoke OUTPUT to reconstruct the output $\boldsymbol{x}$ to Client_i.

Client_i on behalf of honest parties.

In Step 4, $\mathcal{S}$ receives from Client_i the shares of $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$ of honest parties. $\mathcal{S}$ checks whether the shares of $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$ held by honest parties form a valid degree-$(k-1)$ packed Shamir sharing. If true, $\mathcal{S}$ recovers the whole sharing $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$, reconstructs the secret of $[\boldsymbol{x}+\boldsymbol{r}]_{2k-2}$, and computes $\boldsymbol{x}=(\boldsymbol{x}+\boldsymbol{r})-\boldsymbol{r}$. Otherwise, $\mathcal{S}$ sets the shares of $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$ of corrupted parties to be all 0 and sets $\boldsymbol{x}=\mathbf{0}$. In this case, $\mathcal{S}$ will abort during the verification of degree-$(k-1)$ packed Shamir sharings.

In Step 5, $\mathcal{S}$ computes the shares of $[\![\boldsymbol{x}]\!]_{n-1}$ held by corrupted parties.

In Step 6 of INPUT MAL, $\mathcal{S}$ simulates TRAN-MAL. The simulation is done as follows:

1. In Step 2, $\mathcal{S}$ emulates the ideal functionality $\mathcal{F}_{\text{rand-sharing-mal}}$ and receives the shares of corrupted parties and the additive error $\boldsymbol{d}$. Suppose these two sharings are denoted by $([\boldsymbol{r}\|\mathtt{pos}]_{n-1}, [f(\boldsymbol{r})+\boldsymbol{d}\|\mathtt{pos}']_{n-k})$.
2. In Step 3, $\mathcal{S}$ generates a random degree-$(n-1)$ packed Shamir sharing as $[\boldsymbol{x}+\boldsymbol{r}\|\mathtt{pos}]_{n-1}$ based on the shares held by corrupted parties. $\mathcal{S}$ reconstructs the secret $\boldsymbol{x}+\boldsymbol{r}$ and computes $f(\boldsymbol{x}+\boldsymbol{r})$.
3. In Step 4, $\mathcal{S}$ honestly follows the protocol and learns the shares of $[f(\boldsymbol{x}+\boldsymbol{r})\|\mathtt{pos}']_{2k-2}$ of honest parties. Since it is a degree-$(2k-2)$ packed Shamir sharing, $\mathcal{S}$ recovers the whole sharing by using honest parties' shares. Then $\mathcal{S}$ reconstructs the secret $\overline{f(\boldsymbol{x}+\boldsymbol{r})}$ (to distinguish from the correct secret).
4. In Step 5, $\mathcal{S}$ computes the shares of the resulting sharing held by corrupted parties. Note that the resulting sharing has secret $\overline{f(\boldsymbol{x}+\boldsymbol{r})}-(f(\boldsymbol{r})+\boldsymbol{d})$. On the other hand, the correct secret should be $f(\boldsymbol{x})$. Therefore, effectively, the adversary inserts an additive error

$$\boldsymbol{\delta}=(\overline{f(\boldsymbol{x}+\boldsymbol{r})}-(f(\boldsymbol{r})+\boldsymbol{d}))-f(\boldsymbol{x})=\overline{f(\boldsymbol{x}+\boldsymbol{r})}-f(\boldsymbol{x}+\boldsymbol{r})-\boldsymbol{d}.$$

   Note that $\mathcal{S}$ learns $\overline{f(\boldsymbol{x}+\boldsymbol{r})}, f(\boldsymbol{x}+\boldsymbol{r}), \boldsymbol{d}$. $\mathcal{S}$ computes the error $\boldsymbol{\delta}$.

Let $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2$ be the additive errors in the invocation of TRAN-MAL on $[\boldsymbol{x}]_{n-1}$ and the invocation of TRAN-MAL on $[\boldsymbol{\gamma}*\boldsymbol{x}]_{n-1}$. We associate $(\boldsymbol{\delta}_1, \boldsymbol{\delta}_2)$ with the authenticated sharing $[\![\boldsymbol{x}\|\mathtt{pos}]\!]_{n-k}$.

$\mathcal{S}$ invokes the ideal functionality $\mathcal{F}_{\text{main-mal}}$ and sends to $\mathcal{F}_{\text{main-mal}}$ the input of corrupted parties extracted above. Then $\mathcal{S}$ receives the output of corrupted parties from $\mathcal{F}_{\text{main-mal}}$.

Simulating the Evaluation Phase. In Step 4.(a), $\mathcal{S}$ simulates NETWORK-MAL.

1. The first 5 steps only involve local computation. $\mathcal{S}$ computes the shares of $([\boldsymbol{x}'\|\mathtt{pos}]_{n-1}, [\boldsymbol{\gamma}*\boldsymbol{x}'\|\mathtt{pos}]_{n-1})$ held by corrupted parties. For all $i\in\{1,2,\ldots,\ell_1\}$, each value x'_i is in the authenticated degree-$(n-k)$ packed Shamir sharing $[\![\boldsymbol{x}^{(i)}\|\mathtt{pos}^{(i)}]\!]_{n-k}$. Recall that for every authenticated degree-$(n-k)$ packed Shamir sharing, $\mathcal{S}$ has computed a pair of vectors, which are the additive errors to the secret and its MAC due to the behaviors of corrupted parties. $\mathcal{S}$ extracts the errors associated with the value x'_i and its MAC for all $i\in\{1,2,\ldots,\ell_1\}$. Then, for all $i>\ell_1$, $\mathcal{S}$ sets the errors associated with the value x'_i and its MAC to be 0. In this way, $\mathcal{S}$ obtains two vectors $(\boldsymbol{\delta}'_1, \boldsymbol{\delta}'_2)$, which are the additive errors to the secrets of $([\boldsymbol{x}'\|\mathtt{pos}]_{n-1}, [\boldsymbol{\gamma}*\boldsymbol{x}'\|\mathtt{pos}]_{n-1})$.
2. In Step 6, $\mathcal{S}$ simulates TRAN-MAL as described above, and extracts the additive errors $\boldsymbol{\delta}''_1, \boldsymbol{\delta}''_2$ in the invocation of TRAN-MAL on $[\boldsymbol{x}'\|\mathtt{pos}]_{n-1}$ and the invocation of TRAN-MAL

on $[\boldsymbol{\gamma} * \boldsymbol{x}' \| \mathtt{pos}]_{n-1}$. Then the additive errors associated with the authenticated sharing $[\![\boldsymbol{x}]\!]_{n-k}$ are $(f(\boldsymbol{\delta}'_1) + \boldsymbol{\delta}''_1, f(\boldsymbol{\delta}'_2) + \boldsymbol{\delta}''_2)$.

In Step 4.(b), for each group of addition gates with input sharings $[\![\boldsymbol{x}]\!]_{n-k}$ and $[\![\boldsymbol{y}]\!]_{n-k}$, $\mathcal{S}$ simulates PACKED-ADD-MAL.

1. In Step 2, $\mathcal{S}$ computes the shares of $[\![\boldsymbol{z}]\!]_{n-k}$ held by corrupted parties. The vector of additive errors associated with $[\![\boldsymbol{z}]\!]_{n-k}$ are the sum of the vectors of additive errors associated with $[\![\boldsymbol{x}]\!]_{n-k}$ and $[\![\boldsymbol{y}]\!]_{n-k}$. Suppose the vector of additive errors associated with $[\![\boldsymbol{z}]\!]_{n-k}$ is denoted by $(\boldsymbol{\delta}'_1, \boldsymbol{\delta}'_2)$.
2. In Step 3, $\mathcal{S}$ simulates TRAN-MAL as described above, and extracts the additive errors $\boldsymbol{\delta}''_1, \boldsymbol{\delta}''_2$ in the invocation of TRAN-MAL on $[\boldsymbol{z}]_{n-k}$ and the invocation of TRAN-MAL on $[\boldsymbol{\gamma} * \boldsymbol{z}]_{n-k}$. Then the additive errors associated with the authenticated sharing $[\![\boldsymbol{z} \| \mathtt{pos}]\!]_{n-k}$ are $(\boldsymbol{\delta}'_1 + \boldsymbol{\delta}''_1, \boldsymbol{\delta}'_2 + \boldsymbol{\delta}''_2)$.

For each group of multiplication gates with input sharings $[\![\boldsymbol{x}]\!]_{n-k}$ and $[\![\boldsymbol{y}]\!]_{n-k}$, $\mathcal{S}$ simulates PACKED-MULT-MAL.

1. In Step 2 of PACKED-MULT-MAL, $\mathcal{S}$ computes the shares of $[\boldsymbol{x} + \boldsymbol{a}]_{n-1}, [\boldsymbol{y} + \boldsymbol{b}]_{n-1}$ of corrupted parties and sets the shares of $[\boldsymbol{x} + \boldsymbol{a}]_{n-1}, [\boldsymbol{y} + \boldsymbol{b}]_{n-1}$ of honest parties to be uniform elements. Then $\mathcal{S}$ reconstructs the secrets $\boldsymbol{x} + \boldsymbol{a}$ and $\boldsymbol{y} + \boldsymbol{b}$.
2. In Step 3 of PACKED-MULT-MAL, since the whole sharings $[\boldsymbol{x} + \boldsymbol{a}]_{n-1}, [\boldsymbol{y} + \boldsymbol{b}]_{n-1}$ have been generated, $\mathcal{S}$ honestly follows the protocol. At the end of Step 3 of PACKED-MULT-MAL, $\mathcal{S}$ receives the shares of $[\boldsymbol{x} + \boldsymbol{a}]_{k-1}, [\boldsymbol{y} + \boldsymbol{b}]_{k-1}, [(\boldsymbol{x} + \boldsymbol{a}) \cdot (\boldsymbol{y} + \boldsymbol{b})]_{k-1}$ of honest parties from P_1 (which is either honestly simulated by $\mathcal{S}$ or controlled by the adversary). $\mathcal{S}$ checks whether the shares of $[\boldsymbol{x} + \boldsymbol{a}]_{k-1}, [\boldsymbol{y} + \boldsymbol{b}]_{k-1}, [(\boldsymbol{x} + \boldsymbol{a}) \cdot (\boldsymbol{y} + \boldsymbol{b})]_{k-1}$ of honest parties form valid degree-$(k-1)$ packed Shamir sharings.
   - If true, $\mathcal{S}$ recovers the whole sharings $[\boldsymbol{x} + \boldsymbol{a}]_{k-1}, [\boldsymbol{y} + \boldsymbol{b}]_{k-1}, [(\boldsymbol{x} + \boldsymbol{a}) \cdot (\boldsymbol{y} + \boldsymbol{b})]_{k-1}$ using the shares of honest parties and then reconstructs the secrets $\overline{\boldsymbol{x} + \boldsymbol{a}}, \overline{\boldsymbol{y} + \boldsymbol{b}}$, and $\overline{(\boldsymbol{x} + \boldsymbol{a}) \cdot (\boldsymbol{y} + \boldsymbol{b})}$. Then $\mathcal{S}$ computes three vectors

$$\begin{aligned} \boldsymbol{\eta}^{(1)} &= (\overline{\boldsymbol{x} + \boldsymbol{a}}) - (\boldsymbol{x} + \boldsymbol{a}) \\ \boldsymbol{\eta}^{(2)} &= (\overline{\boldsymbol{y} + \boldsymbol{b}}) - (\boldsymbol{y} + \boldsymbol{b}) \\ \boldsymbol{\eta}^{(3)} &= \overline{(\boldsymbol{x} + \boldsymbol{a}) \cdot (\boldsymbol{y} + \boldsymbol{b})} - (\overline{\boldsymbol{x} + \boldsymbol{a}})) \cdot (\overline{\boldsymbol{y} + \boldsymbol{b}}) \end{aligned}$$

   - Otherwise, $\mathcal{S}$ sets the shares of $[\boldsymbol{x} + \boldsymbol{a}]_{k-1}, [\boldsymbol{y} + \boldsymbol{b}]_{k-1}$ of corrupted parties to be all 0. In this case, $\mathcal{S}$ will abort during the verification of degree-$(k-1)$ packed Shamir sharings.
3. In Step 4 of PACKED-MULT-MAL, $\mathcal{S}$ computes the shares of $[\boldsymbol{z}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{z}]_{n-1}$ of corrupted parties.
4. In Step 5, $\mathcal{S}$ simulates TRAN-MAL as described above, and extracts the additive errors $\boldsymbol{\delta}_1, \boldsymbol{\delta}_2$ in the invocation of TRAN-MAL on $[\boldsymbol{z}]_{n-1}$ and the invocation of TRAN-MAL on $[\boldsymbol{\gamma} * \boldsymbol{z}]_{n-1}$. Then the additive errors associated with the authenticated sharing $[\![\boldsymbol{z} \| \mathtt{pos}]\!]_{n-k}$ are $(\boldsymbol{\delta}_1, \boldsymbol{\delta}_2)$.

Simulating the Output Phase. For every group of k output gates of Client_i, suppose $\boldsymbol{x}$ are the inputs. $\mathcal{S}$ first simulates NETWORK-MAL as described above and compute the additive errors associated with the authenticated degree-$(n-k)$ Shamir sharing $[\![\boldsymbol{x}]\!]_{n-k}$.

In Step 5.(a), $\mathcal{S}$ computes the shares of $[\boldsymbol{x}+\boldsymbol{r}]_{n-1}$ of corrupted parties and sets the shares of $[\boldsymbol{x}+\boldsymbol{r}]_{n-1}$ of honest parties to be uniform elements. Then $\mathcal{S}$ reconstructs the secrets $\boldsymbol{x}+\boldsymbol{r}$.

In Step 5.(b), $\mathcal{S}$ honestly follows the protocol and learns the shares of $[\boldsymbol{x}+\boldsymbol{r}]_{2k-2}$ held by honest parties. $\mathcal{S}$ recovers the whole sharings $[\boldsymbol{x}+\boldsymbol{r}]_{2k-2}$ using the shares of honest parties and then reconstructs the secrets $\overline{\boldsymbol{x}+\boldsymbol{r}}$. Then $\mathcal{S}$ computes $\boldsymbol{\eta} = \overline{\boldsymbol{x}+\boldsymbol{r}} - (\boldsymbol{x}+\boldsymbol{r})$.

Then $\mathcal{S}$ simulates the verification process. For CHECK-CONSISTENCY,

1. In Step 2, for each honest party P_i, $\mathcal{S}$ emulates $\mathcal{F}_{\text{com}}$ by outputting (i, τ_{λ_i}) to all parties. For each corrupted party P_i, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\text{com}}$ and learns λ_i.
2. In Step 3, without loss of generality, assume that P_n is an honest party. $\mathcal{S}$ samples a random element as λ. For each honest party $P_i \neq P_n$, $\mathcal{S}$ samples a random element as λ_i. Then λ_n is set to be $\lambda - \sum_{i=1}^{n-1} \lambda_i$. For each honest party P_i, $\mathcal{S}$ emulates $\mathcal{F}_{\text{com}}$ by outputting $(\lambda_i, i, \tau_{\lambda_i})$. For each corrupted party P_i, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\text{com}}$.
3. Recall that for all degree-$(k-1)$ packed Shamir sharings generated in PACKED-MULT-MAL, $\mathcal{S}$ has already learnt the shares of honest parties. Therefore, $\mathcal{S}$ honestly follows the rest of steps in CHECK-CONSISTENCY. If no party aborts at the end of PACKED-MULT-MAL but there exists a degree-$(k-1)$ packed Shamir sharing such that the shares of honest parties do not form a valid degree-$(k-1)$ packed Shamir sharing, $\mathcal{S}$ aborts.

For CHECK-SECRET,

1. In Step 1, a pair of sharings $([\boldsymbol{u}^{(i)}]_{2k-2}, [\![\boldsymbol{u}^{(i)}]\!]_{n-k})$ may come from three places:
   - It may be $([\boldsymbol{x}+\boldsymbol{a}]_{k-1}, [\![\boldsymbol{x}+\boldsymbol{a}]\!]_{n-k})$ in PACKED-MULT-MAL. In this case, $\mathcal{S}$ has computed the error $\boldsymbol{\eta}^{(1)}$ which is the secret of $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}$ minus the secret of $[\![\boldsymbol{x}+\boldsymbol{a}]\!]_{n-k}$. Note that, $\mathcal{S}$ learns the secrets of both sharings.
   - It may be $([(\boldsymbol{x}+\boldsymbol{a})\cdot(\boldsymbol{y}+\boldsymbol{b})]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1}\cdot[\boldsymbol{y}+\boldsymbol{b}]_{k-1}, 0)$ in PACKED-MULT-MAL, where the second sharing is all-0 sharing. In this case, $\mathcal{S}$ has computed the error $\boldsymbol{\eta}^{(3)}$, which is the secret of $[(\boldsymbol{x}+\boldsymbol{a})\cdot(\boldsymbol{y}+\boldsymbol{b})]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1}\cdot[\boldsymbol{y}+\boldsymbol{b}]_{k-1}$. Note that, $\mathcal{S}$ learns the secrets of both sharings.
   - It may be $([\boldsymbol{x}+\boldsymbol{r}]_{k-1}, [\![\boldsymbol{x}+\boldsymbol{r}]\!]_{n-k})$ in Step 5 of PACKED-MAIN-MAL. In this case, $\mathcal{S}$ has computed the error $\boldsymbol{\eta}$ which is the secret of $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$ minus the secret of $[\![\boldsymbol{x}+\boldsymbol{r}]\!]_{n-k}$. Note that, $\mathcal{S}$ learns the secrets of both sharings.

   Thus, in any case, $\mathcal{S}$ learns the secrets of both sharings. $\mathcal{S}$ also learns the shares held by corrupted parties, and the additive errors associated with $[\![\boldsymbol{u}^{(i)}]\!]_{n-k}$.
2. In Step 2 and Step 3, $\mathcal{S}$ emulates the ideal functionality $\mathcal{F}_{\text{com}}$ in the same way as that in CHECK-CONSISTENCY.
3. In Step 4,
   - $\mathcal{S}$ computes the whole sharing $[\boldsymbol{u}]_{2k-2}$ and the secret $\overline{\boldsymbol{u}}$
   - $\mathcal{S}$ computes the shares of $[\boldsymbol{u}]_{n-1}$ held by corrupted parties, and the secret $\boldsymbol{u}$. Then $\mathcal{S}$ randomly samples the shares of honest parties based on the secret $\boldsymbol{u}$ and the shares of corrupted parties.

- $\mathcal{S}$ samples a random element as γ and sets $\boldsymbol{\gamma} = (\gamma, \gamma, \ldots, \gamma) \in \mathbb{F}^k$. Then $\mathcal{S}$ computes the shares of $[\boldsymbol{\gamma}]_{n-k}$ based on the secret $\boldsymbol{\gamma}$ and the shares of corrupted parties.
- $\mathcal{S}$ computes the shares of $[\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}$ held by corrupted parties. Recall that $\mathcal{S}$ has computed a vector of additive errors for each $[\![\boldsymbol{u}^{(i)}]\!]_{n-k}$, denoted by $(\boldsymbol{\delta}_1^{(i)}, \boldsymbol{\delta}_2^{(i)})$. $\mathcal{S}$ computes the additive errors to $([\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1})$, denoted by $(\boldsymbol{\delta}_1, \boldsymbol{\delta}_2)$. Then $\mathcal{S}$ computes the secret of $[\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}$ by $\overline{\boldsymbol{\gamma} * \boldsymbol{u}} = \boldsymbol{\gamma} * \boldsymbol{u} + \boldsymbol{\delta}_2 - \boldsymbol{\gamma} * \boldsymbol{\delta}_1$. $\mathcal{S}$ randomly samples the shares of $[\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}$ of honest parties based on the secret $\overline{\boldsymbol{\gamma} * \boldsymbol{u}}$ and the shares of corrupted parties.

4. In Step 5 and Step 6, since the whole sharings $[\boldsymbol{u}]_{2k-2}, [\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma}]_{n-k}$ have been generated, $\mathcal{S}$ honestly follows the protocol and honestly emulates $\mathcal{F}_{\text{com}}$. If no party aborts at the end of CHECK-SECRET but the following case occurs, $\mathcal{S}$ aborts.
   - There exists $([\boldsymbol{u}^{(i)}]_{2k-2}, [\![\boldsymbol{u}^{(i)}]\!]_{n-k})$ such that their secrets do not satisfy

$$\bar{\boldsymbol{u}}^{(i)} = \boldsymbol{u}^{(i)} - \boldsymbol{\delta}_1^{(i)}.$$

After the verification, $\mathcal{S}$ simulates OUTPUT-MAL.

1. If Client_i is corrupted, $\mathcal{S}$ learns the output $\boldsymbol{x}$ from $\mathcal{F}_{\text{main-mal}}$. Recall that $\mathcal{S}$ learns the whole sharing of $[\boldsymbol{x} + \boldsymbol{r}]_{2k-2}$ and the secret $\boldsymbol{x} + \boldsymbol{r}$. $\mathcal{S}$ computes $\boldsymbol{r} = (\boldsymbol{x} + \boldsymbol{r}) - \boldsymbol{x}$. Then $\mathcal{S}$ recovers the whole sharing $[\boldsymbol{r}]_{n-k}$ using the secret $\boldsymbol{r}$ and the shares of corrupted parties. $\mathcal{S}$ samples a random vector as $\boldsymbol{\Delta}$ and computes $\boldsymbol{\Delta} * \boldsymbol{r}$. Then $\mathcal{S}$ recovers the whole sharings $[\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ using the secrets $\boldsymbol{\Delta}, \boldsymbol{\Delta} * \boldsymbol{r}$ and the shares of corrupted parties.
   Since the whole sharings $[\boldsymbol{x} + \boldsymbol{r}]_{2k-2}, [\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ have been generated, $\mathcal{S}$ honestly follows the protocol.
2. If Client_i is honest, $\mathcal{S}$ receives the shares of $[\boldsymbol{x} + \boldsymbol{r}]_{2k-2}, [\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ of corrupted parties. For $[\boldsymbol{x} + \boldsymbol{r}]_{2k-2}$, if the shares received from corrupted parties are not equal to the shares that corrupted parties should hold, $\mathcal{S}$ aborts on behalf of Client_i.
   For each sharing $[\boldsymbol{p}]_{n-k} \in \{[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}\}$, $\mathcal{S}$ computes a sharing $[\delta(\boldsymbol{p})]_{n-k}$ which is defined as follows:
   (a) The shares of all honest parties are set to be 0.
   (b) For each corrupted party P_j, the j-th share is equal to the j-th share of $[\boldsymbol{p}]_{n-k}$ received from P_j minus the same share that P_j should hold.
   $\mathcal{S}$ checks whether $[\delta(\boldsymbol{p})]_{n-k}$ is a valid degree-$(n-k)$ packed Shamir sharing of $\mathbf{0}$. If not, $\mathcal{S}$ aborts on behalf of Client_i.

This completes the description of the simulator $\mathcal{S}$.

Hybrid Argument. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties with overwhelming probability. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ simulates CHECK-CONSISTENCY. Compared with **Hybrid**$_0$, there are two changes:

- The functionality $\mathcal{F}_{\text{com}}$ is emulated by $\mathcal{S}$ and the random element λ is chosen by $\mathcal{S}$. Note that in **Hybrid**$_0$, corrupted parties only receive from $\mathcal{F}_{\text{com}}$ (i, τ_{λ_i}) for each honest party in

Step 2 of CHECK-CONSISTENCY. Therefore, for each corrupted party P_i, the value λ_i is independent of the values of honest parties. It means that the element λ computed in Step 3 of CHECK-CONSISTENCY is uniformly random. In $\mathbf{Hybrid}_1$, $\mathcal{S}$ simply outputs (i, τ_{λ_i}) on behalf of $\mathcal{F}_{\text{com}}$ for each honest party in Step 2 of CHECK-CONSISTENCY. Then $\mathcal{S}$ samples a random element λ and then samples the value λ_i for each honest party such that the summation of all λ_i's is λ. Therefore, the distribution of these two steps is identical to that in $\mathbf{Hybrid}_0$.

- $\mathcal{S}$ will abort if no party aborts in CHECK-CONSISTENCY but there exists a degree-$(k-1)$ packed Shamir sharing such that the shares of honest parties do not form a valid degree-$(k-1)$ packed Shamir sharing. By Lemma 14.2, it happens with probability at most $m/|\mathbb{F}|$.

Recall that $|\mathbb{F}| \geq 2^{\kappa}$, where κ is the security parameter. Therefore, the distribution of $\mathbf{Hybrid}_1$ is statistically close to the distribution of $\mathbf{Hybrid}_0$.

$\mathbf{Hybrid}_2$: In this hybrid, $\mathcal{S}$ computes the shares that corrupted parties should hold as described above. For each authenticated degree-$(n-k)$ packed Shamir sharing, $\mathcal{S}$ also computes a pair of vectors, which are the additive errors to the secret and its MAC. For each degree-$(2k-2)$ (including degree-$(k-1)$) packed Shamir sharing, $\mathcal{S}$ computes the whole sharing by using the shares of honest parties. Note that we do not change the way that $\mathcal{S}$ simulates the behaviors of honest parties. Therefore the distribution of $\mathbf{Hybrid}_2$ is identical to the distribution of $\mathbf{Hybrid}_1$.

$\mathbf{Hybrid}_3$: In this hybrid, $\mathcal{S}$ simulates CHECK-SECRET (including the simulation of the authentication key). Compared with $\mathbf{Hybrid}_2$, there are three changes:

- The functionality $\mathcal{F}_{\text{com}}$ is simulated by $\mathcal{S}$ and the random element λ is chosen by $\mathcal{S}$. Following the same argument as that in $\mathbf{Hybrid}_1$, the distribution of these two steps is identical to that in $\mathbf{Hybrid}_2$.
- $\mathcal{S}$ samples a random element as γ and sets $\boldsymbol{\gamma} = (\gamma, \gamma, \ldots, \gamma)$. Then $\mathcal{S}$ computes the shares of $[\boldsymbol{\gamma}]_{n-k}$ of honest parties based on the secret and the shares of corrupted parties.
  In $\mathbf{Hybrid}_2$, we argue that the messages that honest parties send to corrupted parties before CHECK-SECRET are independent of the shares of $[\boldsymbol{\gamma}]_{n-k}$ held by honest parties. The messages sent from honest parties to corrupted parties have 3 different kinds:
  - The shares of a random degree-$(n-k)$ packed Shamir sharing of honest parties directly learnt from $\mathcal{F}_{\text{prep-mal}}$. This kind includes Step 2 of INPUT-MAL.
  - The shares of a random degree-$(n-1)$ packed Shamir sharing of honest parties, which are uniformly random field elements. This kind includes Step 3 of TRAN-MAL, the first half of Step 3 of PACKED-MULT-MAL, and Step 5.(a) of PACKED-MAIN-MAL.
  - The degree-$(2k-2)$ (including degree-$(k-1)$) packed Shamir sharings whose secrets are uniformly random. This kind includes Step 4 of TRAN-MAL, Step 4 of INPUT-MAL, the second half of Step 3 of PACKED-MULT-MAL, and Step 5.(b) of PACKED-MAIN-MAL.
  - The linear combinations of shares of degree-$(k-1)$ packed Shamir sharings that are directly received from other parties. This kind includes Step 5 of CHECK-CONSISTENCY.

  One can verify that all above messages are independent of the shares of $[\boldsymbol{\gamma}]_{n-k}$ held by honest parties. Since γ is uniformly random given the shares of $[\boldsymbol{\gamma}]_{n-k}$ held by corrupted

parties, the distribution of the shares of $[\boldsymbol{\gamma}]_{n-k}$ held by honest parties is identical in both $\mathbf{Hybrid}_2$ and $\mathbf{Hybrid}_3$.

- For $[\boldsymbol{u}]_{2k-2}$, $\mathcal{S}$ has already computed the whole sharing. Let $\bar{\boldsymbol{u}}$ denote the secret of $[\boldsymbol{u}]_{2k-2}$. For $([\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1})$, the secrets can be computed from the secrets and the MACs of $\{[\![\boldsymbol{u}^{(i)}]\!]_{n-k}\}_{i=1}^{m}$. For each $[\![\boldsymbol{u}^{(i)}]\!]_{n-k}$, $\mathcal{S}$ computes the secret by using the shares that corrupted parties should hold and the shares of honest parties. Later on, we will show that $\mathcal{S}$ can compute the secret $\boldsymbol{u}^{(i)}$ without the shares of honest parties. Note that $\mathcal{S}$ also computes a pair of vectors $(\boldsymbol{\delta}_1^{(i)}, \boldsymbol{\delta}_2^{(i)})$ which are additive errors to the secret and its MAC of $[\![\boldsymbol{u}^{(i)}]\!]_{n-k}$. Then, the secret of $[\boldsymbol{\gamma} * \boldsymbol{u}^{(i)}]_{n-k}$ is $\overline{\boldsymbol{\gamma} * \boldsymbol{u}^{(i)}} = \boldsymbol{\gamma} * \boldsymbol{u}^{(i)} + \boldsymbol{\delta}_2^{(i)} - \boldsymbol{\gamma} * \boldsymbol{\delta}_1^{(i)}$. $\mathcal{S}$ computes the secret of $[\boldsymbol{\gamma} * \boldsymbol{u}^{(i)}]_{n-k}$. Now $\mathcal{S}$ can compute the secrets of $([\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1})$. Given the secrets of $[\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}$ and the shares of corrupted parties, $\mathcal{S}$ randomly samples the shares of honest parties.
  In $\mathbf{Hybrid}_2$, for $[\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}$, since all parties use random degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$, $[\boldsymbol{o}^{(1)}]_{n-1}, [\boldsymbol{o}^{(2)}]_{n-1}$, as random masks, they are random degree-$(n-1)$ packed Shamir sharings given the secrets $\boldsymbol{u}, \overline{\boldsymbol{\gamma} * \boldsymbol{u}}$ and the shares of corrupted parties. Note that $\boldsymbol{u}$ is identical to that in $\mathbf{Hybrid}_3$, and $\overline{\boldsymbol{\gamma} * \boldsymbol{u}}$ is determined by $\boldsymbol{u}$ and the pair of additive errors. Therefore, $\boldsymbol{u}, \overline{\boldsymbol{\gamma} * \boldsymbol{u}}$ are identical to those in $\mathbf{Hybrid}_3$. The shares of $[\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}$ of honest parties have the same distribution in both $\mathbf{Hybrid}_2$ and $\mathbf{Hybrid}_3$.
- If no party aborts at the end of CHECK-SECRET but the following case occurs, $\mathcal{S}$ aborts.
  - There exists $([\boldsymbol{u}^{(i)}]_{2k-2}, [\![\boldsymbol{u}^{(i)}]\!]_{n-k})$ such that their secrets do not satisfy

$$\bar{\boldsymbol{u}}^{(i)} = \boldsymbol{u}^{(i)} - \boldsymbol{\delta}_1^{(i)}.$$

  Let $(\boldsymbol{\delta}_1, \boldsymbol{\delta}_2)$ be the pair of additive errors to $[\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}$ computed from $\{(\boldsymbol{\delta}_1^{(i)}, \boldsymbol{\delta}_2^{(i)})\}_{i=1}^{m}$. Note that
  - Following the same argument as that in Lemma 14.2, if there exists a pair $([\boldsymbol{u}^{(i)}]_{2k-2}, [\![\boldsymbol{u}^{(i)}]\!]_{n-k})$ such that $\bar{\boldsymbol{u}}^{(i)} \neq \boldsymbol{u}^{(i)} - \boldsymbol{\delta}_1^{(i)}$, then with overwhelming probability, for the final pair $([\boldsymbol{u}]_{2k-2}, [\![\boldsymbol{u}]\!]_{n-1})$, $\bar{\boldsymbol{u}} \neq \boldsymbol{u} - \boldsymbol{\delta}_1$.
  - Corrupted parties cannot change their shares of $[\boldsymbol{u}]_{2k-2}$ without being detected since the whole sharing is determined by the shares of honest parties.
  - While corrupted parties may change their shares of $[\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma}]_{n-k}$, any change of the shares of corrupted parties either will be detected or is equivalent to additive errors to the secrets $\boldsymbol{u}, \overline{\boldsymbol{\gamma} * \boldsymbol{u}}, \boldsymbol{\gamma}$. To see this, for each of the sharing $[\boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma}]_{n-k}$, we may compute the difference between the sharing that uses the shares that corrupted parties commit and the sharing that uses the shares that corrupted parties should hold. For $[\boldsymbol{u}]_{n-1}$ or $[\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}$, the difference corresponds to a degree-$(n-1)$ packed Shamir sharing of the additive error. For $[\boldsymbol{\gamma}]_{n-k}$, the difference is either not a valid degree-$(n-k)$ packed Shamir sharing, which will be detected when checking $[\boldsymbol{\gamma}]_{n-k}$, or a degree-$(n-k)$ packed Shamir sharing of the additive error. Note that the additive errors are independent of $\boldsymbol{\gamma}, \overline{\boldsymbol{\gamma} * \boldsymbol{u}}$ since the adversary needs to decide the additive errors before reconstructing $[\boldsymbol{\gamma} * \boldsymbol{u}]_{n-1}, [\boldsymbol{\gamma}]_{n-k}$. When $\overline{\boldsymbol{u}} \neq \boldsymbol{u} - \boldsymbol{\delta}_1$, the only possibility that no party aborts at the end of CHECK-SECRET is that the adversary adds $\boldsymbol{\epsilon}_0, \boldsymbol{\epsilon}_1, \boldsymbol{\epsilon}_2$ to $\boldsymbol{u}, \overline{\boldsymbol{\gamma} * \boldsymbol{u}}, \boldsymbol{\gamma}$ respectively such that (1)

$\overline{\boldsymbol{u}} = \boldsymbol{u} + \boldsymbol{\epsilon}_0$, and (2) $(\boldsymbol{\gamma} + \boldsymbol{\epsilon}_2) * (\boldsymbol{u} + \boldsymbol{\epsilon}_0) = \overline{\boldsymbol{\gamma} * \boldsymbol{u}} + \boldsymbol{\epsilon}_1$. Recall that $\overline{\boldsymbol{\gamma} * \boldsymbol{u}} = \boldsymbol{\gamma} * \boldsymbol{u} + \boldsymbol{\delta}_2 - \boldsymbol{\gamma} * \boldsymbol{\delta_1}$. From (1), $\boldsymbol{\epsilon}_0 \neq -\boldsymbol{\delta}_1$. From (2), $\boldsymbol{\gamma} * (\boldsymbol{\epsilon}_0 + \boldsymbol{\delta}_1) = \boldsymbol{\delta}_2 + \boldsymbol{\epsilon}_1 - \boldsymbol{u} * \boldsymbol{\epsilon}_2 - \boldsymbol{\epsilon}_0 * \boldsymbol{\epsilon}_2$. Since γ is a random field element and $\boldsymbol{\gamma} = (\gamma, \gamma, \ldots, \gamma) \in \mathbb{F}^k$, the probability that the adversary can find $\boldsymbol{\epsilon}_0, \boldsymbol{\epsilon}_1, \boldsymbol{\epsilon}_2$ that satisfy (1) and (2) is negligible.

In Summary, the distribution of $\mathbf{Hybrid}_3$ is statistically close to the distribution of $\mathbf{Hybrid}_2$. Note that in $\mathbf{Hybrid}_3$, $\mathcal{S}$ will always abort if the computation is incorrect.

$\mathbf{Hybrid}_4$: In this hybrid, $\mathcal{S}$ extracts the input of corrupted parties as described above. Then $\mathcal{S}$ invokes $\mathcal{F}_{\text{main-mal}}$ and sends the input of corrupted parties to $\mathcal{F}_{\text{main-mal}}$. $\mathcal{S}$ simulates OUTPUT. We first consider the case that the output gates belong to an honest client. Compared with $\mathbf{Hybrid}_3$, the change is the following:

- In $\mathbf{Hybrid}_4$, after $\mathcal{S}$ receives the shares of $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ of corrupted parties, for each sharing, $\mathcal{S}$ computes the difference of the sharing that uses the shares that corrupted send to $\mathcal{S}$ and the sharing that uses the shares that corrupted parties should hold. $\mathcal{S}$ aborts if the difference is not a valid degree-$(n-k)$ packed Shamir sharing of $\mathbf{0}$.
- In $\mathbf{Hybrid}_3$, Client reconstructs the secret of $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ and checks whether the third secret is equal to the coordinate-wise multiplication of the first two secrets. Following the same argument as that in $\mathbf{Hybrid}_3$, the change of the shares of $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}$, $[\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ of corrupted parties either will lead to invalid degree-$(n-k)$ packed Shamir sharings or is equivalent to additive errors to the secrets $\boldsymbol{r}, \boldsymbol{\Delta}, \boldsymbol{\Delta} * \boldsymbol{r}$. If the adversary changes the shares of corrupted parties but Client does not abort, the only possibility is that the adversary adds $\boldsymbol{\epsilon}_0, \boldsymbol{\epsilon}_1, \boldsymbol{\epsilon}_2$ to $\boldsymbol{r}, \boldsymbol{\Delta}, \boldsymbol{\Delta} * \boldsymbol{r}$ respectively such that $(\boldsymbol{\Delta} + \boldsymbol{\epsilon}_1) * (\boldsymbol{r} + \boldsymbol{\epsilon}_0) = \boldsymbol{\Delta} * \boldsymbol{r} + \boldsymbol{\epsilon}_2$. It means that $\boldsymbol{\Delta} * \boldsymbol{\epsilon}_0 + \boldsymbol{\epsilon}_1 * \boldsymbol{r} + \boldsymbol{\epsilon}_1 * \boldsymbol{\epsilon}_0 = \boldsymbol{\epsilon}_2$. If $\boldsymbol{\epsilon}_0 = \boldsymbol{\epsilon}_1 = \boldsymbol{\epsilon}_2 = \mathbf{0}$, then in both $\mathbf{Hybrid}_3$ and $\mathbf{Hybrid}_4$, Client (or $\mathcal{S}$) accepts. If $\boldsymbol{\epsilon}_0 = \mathbf{0}$ but $\boldsymbol{\epsilon}_1 \neq \mathbf{0}$ or $\boldsymbol{\epsilon}_2 \neq \mathbf{0}$, then $\boldsymbol{\epsilon}_1$ and $\boldsymbol{\epsilon}_2$ should satisfy that $\boldsymbol{\epsilon}_1 * \boldsymbol{r} = \boldsymbol{\epsilon}_2$. Since $\boldsymbol{r}$ is a random vector, the probability that the adversary can find such $\boldsymbol{\epsilon}_1$ and $\boldsymbol{\epsilon}_2$ is negligible. Similarly, if $\boldsymbol{\epsilon}_0 \neq \mathbf{0}$, since $\boldsymbol{\Delta}$ is a random vector, the probability that the adversary can find $\boldsymbol{\epsilon}_0, \boldsymbol{\epsilon}_1, \boldsymbol{\epsilon}_2$ that satisfy the above requirement is negligible.

Thus, when the client is honest, the distribution of $\mathbf{Hybrid}_4$ is statistically close to $\mathbf{Hybrid}_3$.

Now we consider the case that the output gates belong to a corrupted client. Compared with $\mathbf{Hybrid}_3$, the change is that $\mathcal{S}$ computes $\boldsymbol{r}$ from $\boldsymbol{x} + \boldsymbol{r}$ and $\boldsymbol{x}$, and $\mathcal{S}$ samples a random vector as $\boldsymbol{\Delta}$. Then based on the secrets and the shares of corrupted parties, $\mathcal{S}$ recovers the whole sharings $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$. Note that OUTPUT is only invoked after CHECK-CONSISTENCY and CHECK-SECRET. Recall that we have changed these protocols by the simulation of $\mathcal{S}$, which will abort if the computation is incorrect. Thus, $\boldsymbol{x}$ received from $\mathcal{F}_{\text{main-mal}}$ is the same as $\boldsymbol{x}$ in $\mathbf{Hybrid}_3$. Therefore, the vector $\boldsymbol{r}, \boldsymbol{\Delta}$ computed by $\mathcal{S}$ have the same distribution as those in $\mathbf{Hybrid}_3$. Since the sharings $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ are determined by the secrets and the shares of corrupted parties, the distribution of $\mathbf{Hybrid}_4$ is identical to $\mathbf{Hybrid}_3$ when the client is corrupted.

In summary, the distribution of $\mathbf{Hybrid}_4$ is statistically close to the distribution of $\mathbf{Hybrid}_3$.

$\mathbf{Hybrid}_5$: In this hybrid, $\mathcal{S}$ simulates TRAN-MAL, PACKED-MULT-MAL, PACKED-ADD-MAL, and Step 5.(a), 5.(b) of PACKED-MULT-MAL. The only difference is that when all parties need to send a degree-$(n-1)$ packed Shamir sharing to P_1, $\mathcal{S}$ samples random elements as the

shares of honest parties. Recall that we either use a random degree-$(n-1)$ packed Shamir sharing $[\boldsymbol{r}]_{n-1}$, or use a pair of random sharings $([\boldsymbol{r}]_{n-k}, [\boldsymbol{o}]_{n-1})$ where $\boldsymbol{o} = \mathbf{0}$, as a random mask. Note that $[\boldsymbol{r}]_{n-k} + [\boldsymbol{o}]_{n-1}$ has the same distribution as $[\boldsymbol{r}]_{n-1}$. Therefore, the sharings that P_1 collects from all parties are always random degree-$(n-1)$ packed Shamir sharings. Given the shares of corrupted parties, the shares of a random degree-$(n-1)$ packed Shamir sharing of honest parties are uniformly random. Thus, the distribution of $\mathbf{Hybrid}_5$ is identical to the distribution of $\mathbf{Hybrid}_4$. Note that the shares generated for honest parties together with the shares of corrupted parties can be used to compute the secret of $[\![\boldsymbol{u}^{(i)}]\!]_{n-k}$ in CHECK-SECRET.

$\mathbf{Hybrid}_6$: In this hybrid, $\mathcal{S}$ simulates INPUT. We first consider the case that the input gates belong to an honest client. Compared with $\mathbf{Hybrid}_5$, the difference is the following:

- In $\mathbf{Hybrid}_6$, after $\mathcal{S}$ receives the shares of $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ of corrupted parties, for each sharing, $\mathcal{S}$ computes the difference of the sharing that uses the shares that corrupted parties send to $\mathcal{S}$ and the sharing that uses the shares that corrupted parties should hold. If the difference is not a valid degree-$(n-k)$ packed Shamir sharing of $\mathbf{0}$, $\mathcal{S}$ aborts. Otherwise $\mathcal{S}$ generates a random vector as $\boldsymbol{x} + \boldsymbol{r}$ and honestly follows the protocol by generating a random degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{x} + \boldsymbol{r}]_{k-1}$ and distributing the shares to corrupted parties.
- In $\mathbf{Hybrid}_5$, Client reconstructs the secret of $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ and checks whether the third secret is equal to the coordinate-wise multiplication of the first two secrets. Following the same argument as that in $\mathbf{Hybrid}_4$, if corrupted parties change their shares of an authenticated sharing (which causes the the change of the secret), with overwhelming probability, the change will be detected and Client will abort. If Client accepts, Client will generate and distribute a random degree-$(k-1)$ packed Shamir sharing of $\boldsymbol{x} + \boldsymbol{r}$. Since $\boldsymbol{r}$ is a random vector, $\boldsymbol{x} + \boldsymbol{r}$ is also a random vector.

Thus, when the client is honest, the distribution of $\mathbf{Hybrid}_6$ is statistically close to $\mathbf{Hybrid}_5$.

Now we consider the case that the input gates belong to a corrupted client. Compared with $\mathbf{Hybrid}_5$, the change is that $\mathcal{S}$ samples two random vector as $\boldsymbol{r}, \boldsymbol{\Delta}$. Then based on the secrets and the shares of corrupted parties, $\mathcal{S}$ recovers the whole sharings $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$. Note that $\boldsymbol{r}, \boldsymbol{\Delta}$ are indeed random vectors in $\mathbf{Hybrid}_5$. Since the sharings $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\Delta}]_{n-k}, [\boldsymbol{\Delta} * \boldsymbol{r}]_{n-k}$ are determined by the secrets and the shares of corrupted parties, the distribution of $\mathbf{Hybrid}_6$ is identical to $\mathbf{Hybrid}_5$ when the client is corrupted.

In summary, the distribution of $\mathbf{Hybrid}_6$ is statistically close to the distribution of $\mathbf{Hybrid}_5$.

Note that $\mathbf{Hybrid}_6$ is the execution in the ideal world, and the distribution of $\mathbf{Hybrid}_6$ is statistically close to the distribution of $\mathbf{Hybrid}_0$, the execution in the real world. □

Theorem 14.1. *In the client-server model, let* c *denote the number of clients,* n *denote the number of parties (servers), and* t *denote the number of corrupted parties (servers). Let* κ *be the security parameter and* $\mathbb{F}$ *be a finite field of size* $|\mathbb{F}| \geq 2^{\kappa}$*. For an arithmetic circuit* C *over* $\mathbb{F}$ *and for all* $\frac{n-1}{3} \leq t \leq n-1$*, there exists an information-theoretic MPC protocol in the circuit-independent preprocessing model which securely computes the arithmetic circuit* C *in the presence of a fully malicious adversary controlling up to* c *clients and* t *parties. The cost of the protocol is* $O(|C| \cdot \frac{n^2}{k^2} + \mathit{poly}(\mathsf{Depth}, \mathsf{c}, n))$ *field elements of preprocessing data and*

$O(|C| \cdot \frac{n}{k} + \textit{poly}(\mathsf{Depth}, \mathsf{c}, n))$ *field elements of communication, where* $k = \frac{n-t+1}{2}$ *and* Depth *is the circuit depth.*

Chapter 15

Honest Majority MPC with Sublinear Online Communication

In this chapter, we focus on the malicious security in the standard honest majority setting where the number of corrupted parties $t = (n-1)/2$. Recall that in our setting, the number of corrupted parties is $n - 2k + 1$. The honest majority setting corresponds to the case where $k = (n+3)/4$ (assuming that $n \equiv 1 \mod 4$). Relying on our efficient multiplication protocol constructed in Chapter 8.1 and efficient multiplication verification protocol constructed in Chapter 7, we show how to efficiently realize $\mathcal{F}_{\text{prep-mal}}$ in the information-theoretical setting. Combining with our protocol for the online phase, we obtain an information-theoretic n-party MPC protocol which securely computes a single arithmetic circuit in the presence of a malicious adversary controlling up to $t = (n-1)/2$ parties with offline communication complexity $O(|C|n)$ and online communication complexity $O(|C|)$ among all parties. Note that the online communication is sublinear in the number of parties. To the best of our knowledge, this is the first work that achieves sublinear online communication complexity in the number of parties in the information-theoretic setting with honest majority.

We first review a protocol from [62] that efficiently prepares a batch of random sharings with malicious security.

15.1 Preparing Random Sharings in Batch

For a general $\mathbb{F}$-arithmetic secret sharing scheme Σ, let $[\![x]\!]$ denote a sharing in Σ of secret x.

We first introduce a tensoring-up lemma from [21].

A Tensoring-up Lemma. We follow the definition of interleaved arithmetic secret sharing scheme: the m-fold interleaved arithmetic secret sharing scheme $\Sigma^{\times m}$ is an n-party scheme which corresponds to m Σ-sharings. We have the following proposition from [21]:

Proposition 15.1 ([21])**.** *Let $\mathbb{K}$ be a degree-m extension field of $\mathbb{F}$ and let Σ be an $\mathbb{F}$-arithmetic secret sharing scheme. Then the m-fold interleaved $\mathbb{F}$-arithmetic secret sharing scheme $\Sigma^{\times m}$ is naturally viewed as a $\mathbb{K}$-arithmetic secret sharing scheme, compatible with its $\mathbb{F}$-linearity.*

This proposition allows us to define $\lambda : \Sigma^{\times m} \to \Sigma^{\times m}$ for every $\lambda \in \mathbb{K}$ such that for all $[\![\boldsymbol{x}]\!] = ([\![x_1]\!], \ldots, [\![x_m]\!]) \in \Sigma^{\times m}$:

- for all $\lambda \in \mathbb{F}$, $\lambda \cdot ([\![x_1]\!], \ldots, [\![x_m]\!]) = (\lambda \cdot [\![x_1]\!], \ldots, \lambda \cdot [\![x_m]\!])$;
- for all $\lambda_1, \lambda_2 \in \mathbb{K}$, $\lambda_1 \cdot [\![\boldsymbol{x}]\!] + \lambda_2 \cdot [\![\boldsymbol{x}]\!] = (\lambda_1 + \lambda_2) \cdot [\![\boldsymbol{x}]\!]$;
- for all $\lambda_1, \lambda_2 \in \mathbb{K}$, $\lambda_1 \cdot (\lambda_2 \cdot [\![\boldsymbol{x}]\!]) = (\lambda_1 \cdot \lambda_2) \cdot [\![\boldsymbol{x}]\!]$.

An Example of an Arithmetic Secret Sharing Scheme and Using the Tensoring-up Lemma. We will use the standard Shamir secret sharing scheme as an example of an arithmetic secret sharing scheme and show how to use the tensoring-up lemma. For a field $\mathbb{F}$ (of size $|\mathbb{F}| \geq n+1$), we may define a secret sharing Σ which takes an input $x \in \mathbb{F}$ and outputs $[x]_t$, i.e., a degree-t Shamir sharing.

A sharing $([x_1]_t, [x_2]_t, \ldots, [x_m]_t) \in \Sigma^{\times m}$ is a vector of m sharings in Σ. Let $\mathbb{K}$ be a degree-m extension field of $\mathbb{F}$. The tensoring-up lemma says that $\Sigma^{\times m}$ is a $\mathbb{K}$-arithmetic secret sharing scheme. Therefore we can perform $\mathbb{K}$-linear operations to the sharings in $\Sigma^{\times m}$.

A High-level Idea of the Construction. As [62], we will follow the idea in [10] of preparing random degree-t Shamir sharings to prepare random sharings in Σ. At a high-level, each party first deals a batch of random sharings in Σ. For each party, all parties together verify that the sharings dealt by this party have the correct form. Then all parties locally convert the sharings dealt by each party to random sharings such that the secrets are not known to any single party.

Recall that κ denotes the security parameter. Let $\mathbb{K}$ be an extension field of $\mathbb{F}$ such that $|\mathbb{K}| \geq 2^\kappa$. Let $m = [\mathbb{K} : \mathbb{F}]$ denote the degree of the extension. Then m is bounded by κ. In the following, instead of preparing random sharings in Σ, we choose to prepare random sharings in $\Sigma^{\times m}$, where each sharing $[\![\boldsymbol{x}]\!]$ in $\Sigma^{\times m}$ is a vector of m sharings $([\![x_1]\!], [\![x_2]\!], \ldots, [\![x_m]\!])$ in Σ. According to Proposition 15.1, $\Sigma^{\times m}$ is an $\mathbb{K}$-arithmetic secret sharing scheme.

15.1.1 Preparing Verified Sharings.

The first step is to let each party deal a batch of random sharings in $\Sigma^{\times m}$. The protocol VERSHARE(P_d, N') (Protocol 15.1) allows the dealer P_d to deal N' random sharings in $\Sigma^{\times m}$. Suppose the share size of a sharing in Σ is ℓ field elements in $\mathbb{F}$. Then the share size of a sharing in $\Sigma^{\times m}$ is $m \cdot \ell$ field elements in $\mathbb{F}$. Recall that the communication complexity of the instantiation of $\mathcal{F}_{\text{coin}}(\mathbb{K})$ (See Chapter 5.4) is $O(n^2)$ elements in $\mathbb{K}$, which is $O(n^2 \cdot m)$ elements in $\mathbb{K}$. The communication complexity of VERSHARE(P_d, N') is $O(N' \cdot n \cdot m \cdot \ell + n^2 \cdot m)$ elements in $\mathbb{F}$.

For a nonempty set A, let

$$\Sigma(A, (a_i)_{i \in A}) := \{[\![x]\!] \mid [\![x]\!] \in \Sigma \text{ and } \pi_A([\![x]\!]) = (a_i)_{i \in A}\}.$$

For a sharing $[\![s]\!]$, we say that $\pi_A([\![s]\!])$ is valid if $\Sigma(A, \pi_A([\![s]\!]))$ is nonempty. For a sharing $[\![\boldsymbol{s}]\!]$ and a nonempty set A, we say $\pi_A([\![\boldsymbol{s}]\!])$ is valid if $\Sigma^{\times m}(A, \pi_A([\![\boldsymbol{s}]\!]))$ (which can be similarly defined) is nonempty.

Lemma 15.1. *Let $\mathcal{H} \subset \mathcal{I}$ denote the set of all honest parties. If all parties accept the verification in the last step of* VERSHARE*(P_d, N'), then the probability that there exists $[\![\boldsymbol{s}^\star]\!] \in \{[\![\boldsymbol{s}^{(\ell)}]\!]\}_{\ell=1}^{N'}$ such that $\pi_{\mathcal{H}}([\![\boldsymbol{s}^\star]\!])$ is invalid is bounded by $N'/2^\kappa$.*

Figure 15.1: Protocol VERSHARE(P_d, N')

1. For $\ell = 1, 2, \ldots, N'$, P_d randomly samples a sharing $[\![\boldsymbol{s}^{(\ell)}]\!]$ in $\Sigma^{\times m}$. P_d distributes the shares of $[\![\boldsymbol{s}^{(\ell)}]\!]$ to all other parties.
2. P_d randomly samples a sharing $[\![\boldsymbol{s}^{(0)}]\!]$ in $\Sigma^{\times m}$. P_d distributes the shares of $[\![\boldsymbol{s}^{(0)}]\!]$ to all other parties. This sharing is used as a random mask when verifying the random sharings generated in Step 1.
3. All parties invoke $\mathcal{F}_{\text{coin}}(\mathbb{K})$ and receive a random field element $\alpha \in \mathbb{K}$. Then, all parties locally compute
$$[\![\boldsymbol{s}]\!] := [\![\boldsymbol{s}^{(0)}]\!] + \alpha \cdot [\![\boldsymbol{s}^{(1)}]\!] + \alpha^2 \cdot [\![\boldsymbol{s}^{(2)}]\!] + \ldots + \alpha^{N'} \cdot [\![\boldsymbol{s}^{(N')}]\!].$$
4. Each party P_j sends the j-th share of $[\![\boldsymbol{s}]\!]$ to all other parties. Then, each party P_j verifies that $[\![\boldsymbol{s}]\!]$ is a valid sharing in $\Sigma^{\times m}$. If not, P_j aborts. Otherwise, all parties take $\{[\![\boldsymbol{s}^{(\ell)}]\!]\}_{\ell=1}^{N'}$ as output.

Proof. Suppose that there exists $[\![\boldsymbol{s}^\star]\!] \in \{[\![\boldsymbol{s}^{(\ell)}]\!]\}_{\ell=1}^{N'}$ such that $\pi_{\mathcal{H}}([\![\boldsymbol{s}^\star]\!])$ is invalid. Now we show that the number of $\alpha \in \mathbb{K}$ such that $[\![\boldsymbol{s}]\!]$ passes the verification in the last step of VERSHARE(P_d, N') is bounded by N'. Then, the lemma follows from that α output by $\mathcal{F}_{\text{coin}}$ is uniformly random in $\mathbb{K}$ and $|\mathbb{K}| \geq 2^\kappa$. Note that if $[\![\boldsymbol{s}]\!]$ passes the verification, then $\pi_{\mathcal{H}}([\![\boldsymbol{s}]\!])$ is valid since $[\![\boldsymbol{s}]\!] \in \Sigma^{\times m}(\mathcal{H}, \pi_{\mathcal{H}}([\![\boldsymbol{s}]\!]))$.

Now assume that there are $N'+1$ different values $\alpha_0, \alpha_1, \ldots, \alpha_{N'}$ such that for all $i \in \{0, 1, \ldots N'\}$,
$$[\![\boldsymbol{s}'_i]\!] := [\![\boldsymbol{s}^{(0)}]\!] + \alpha_i \cdot [\![\boldsymbol{s}^{(1)}]\!] + \alpha_i^2 \cdot [\![\boldsymbol{s}^{(2)}]\!] + \ldots + \alpha_i^{N'} \cdot [\![\boldsymbol{s}^{(N')}]\!].$$
can pass the verification, which implies that for all $i \in \{0, 1, \ldots, N'\}$, $\pi_{\mathcal{H}}([\![\boldsymbol{s}'_i]\!])$ is valid. Let $\boldsymbol{M}$ be a matrix of size $(N'+1) \times (N'+1)$ in $\mathbb{K}$ such that $\boldsymbol{M}_{ij} = \alpha_{i-1}^{j-1}$. Then we have
$$([\![\boldsymbol{s}'_0]\!], [\![\boldsymbol{s}'_1]\!], \ldots, [\![\boldsymbol{s}'_{N'}]\!])^{\mathrm{T}} = \boldsymbol{M} \cdot ([\![\boldsymbol{s}^{(0)}]\!], [\![\boldsymbol{s}^{(1)}]\!], \ldots, [\![\boldsymbol{s}^{(N')}]\!])^{\mathrm{T}}.$$
Note that $\boldsymbol{M}$ is a $(N'+1) \times (N'+1)$ Vandermonde matrix, which is invertible. Therefore,
$$([\![\boldsymbol{s}^{(0)}]\!], [\![\boldsymbol{s}^{(1)}]\!], \ldots, [\![\boldsymbol{s}^{(N')}]\!])^{\mathrm{T}} = \boldsymbol{M}^{-1} \cdot ([\![\boldsymbol{s}'_0]\!], [\![\boldsymbol{s}'_1]\!], \ldots, [\![\boldsymbol{s}'_{N'}]\!])^{\mathrm{T}}.$$
Since $\Sigma^{\times m}$ is $\mathbb{K}$-linear, and $\pi_{\mathcal{H}}([\![\boldsymbol{s}'_i]\!])$ is valid for all $i \in \{0, 1, \ldots, N'\}$ by assumption, we have that
$$\pi_{\mathcal{H}}([\![\boldsymbol{s}^{(0)}]\!]), \pi_{\mathcal{H}}([\![\boldsymbol{s}^{(1)}]\!]), \ldots, \pi_{\mathcal{H}}([\![\boldsymbol{s}^{(N')}]\!])$$
are all valid, which leads to a contradiction. □

15.1.2 Converting to Random Sharings.

Let $[\![\boldsymbol{s}_d]\!]$ denote a sharing in $\Sigma^{\times m}$ dealt by P_d in VERSHARE. Let $t < n$ denote the number of corrupted parties. We will convert these n sharings, one sharing dealt by each party, to $(n-t)$

random sharings in $\Sigma^{\times m}$. As [10], this is achieved by making use of the fact that the transpose of a Vandermonde matrix acts as a randomness extractor. The description of CONVERT appears in Protocol 15.2.

Figure 15.2: Protocol CONVERT

1. For each party P_d, let $[\![\boldsymbol{s}_d]\!]$ denote a sharing in $\Sigma^{\times m}$ dealt by P_d in VERSHARE. Let $\boldsymbol{M}^{\mathrm{T}}$ be an $n \times (n-t)$ Vandermonde matrix in $\mathbb{K}$. Then $\boldsymbol{M}$ is a matrix of size $(n-t) \times n$.
2. All parties compute
$$([\![\boldsymbol{r}_1]\!], \ldots, [\![\boldsymbol{r}_{n-t}]\!])^{\mathrm{T}} := \boldsymbol{M} \cdot ([\![\boldsymbol{s}_1]\!], \ldots, [\![\boldsymbol{s}_n]\!])^{\mathrm{T}}.$$
3. All parties take $[\![\boldsymbol{r}_1]\!], \ldots, [\![\boldsymbol{r}_{n-t}]\!]$ as output.

Combining VERSHARE and CONVERT, we have BATCH-RAND(N) (Protocol 15.3) which securely computes N copies of $\mathcal{F}_{\text{rand-sharing}}(\Sigma)$ (restated below). We show that this construction is secure for any corruption threshold $t < n$. We will analyze the communication complexity after the proof of Lemma 15.2.

Figure 11.1: Functionality $\mathcal{F}_{\text{rand-sharing}}(\Sigma)$

1. $\mathcal{F}_{\text{rand-sharing}}$ receives the set of corrupted parties, denoted by $\mathcal{C}orr$.
2. $\mathcal{F}_{\text{rand-sharing}}$ receives from the adversary a set of shares $\{\boldsymbol{u}_j\}_{j \in \mathcal{C}orr}$ where $\boldsymbol{u}_j \in \tilde{U}$ for all $j \in \mathcal{C}orr$.
3. $\mathcal{F}_{\text{rand-sharing}}$ samples a random Σ-sharing $\boldsymbol{X}$ such that the shares of $\boldsymbol{X}$ held by corrupted parties are identical to those received from the adversary, i.e., $\pi_{\mathcal{C}orr}(\boldsymbol{X}) = (\boldsymbol{u}_j)_{j \in \mathcal{C}orr}$. If such a sharing does not exist, $\mathcal{F}_{\text{rand-sharing}}$ sends `abort` to all honest parties and halts.
4. Otherwise, $\mathcal{F}_{\text{rand-sharing}}$ distributes the shares of $\boldsymbol{X}$ to honest parties.

Lemma 15.2. *Let $t < n$ be a positive integer. Protocol* BATCH-RAND *securely computes N copies of $\mathcal{F}_{\text{rand-sharing}}$ (Functionality 11.1) with abort in the $\mathcal{F}_{\text{coin}}$-hybrid model in the presence of a malicious adversary who controls t parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}orr$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties.

Simulation for VERSHARE. We first consider the case where P_d is an honest party.

- In Step 1 and Step 2, P_d needs to distribute random sharings $\{[\![\boldsymbol{s}^{(\ell)}]\!]\}_{\ell=0}^{N'}$ in $\Sigma^{\times m}$. For each sharing $[\![\boldsymbol{s}^{(\ell)}]\!]$, $\mathcal{S}$ samples a random sharing $[\![\boldsymbol{s}^{(\ell)}]\!] \in \Sigma^{\times m}$ and sends the shares of corrupted parties $\pi_{\mathcal{C}orr}([\![\boldsymbol{s}^{(\ell)}]\!])$ to $\mathcal{A}$.

Figure 15.3: Protocol BATCH-RAND(N)

1. Let $N' = \frac{N}{m(n-t)}$. For each party P_d, all parties invoke VERSHARE(P_d, N'). Let $\{[\![\boldsymbol{s}_d^{(\ell)}]\!]\}_{\ell=1}^{N'}$ denote the output.
2. For each $\ell \in \{1, \ldots, N'\}$, all parties invoke CONVERT on $\{[\![\boldsymbol{s}_d^{(\ell)}]\!]\}_{d=1}^{n}$. Let $\{[\![\boldsymbol{r}_i^{(\ell)}]\!]\}_{i=1}^{n-t}$ denote the output.
3. For each sharing $[\![\boldsymbol{r}_i^{(\ell)}]\!]$, all parties separate it into m sharings in Σ. Note that there are in total $N' \cdot (n-t) \cdot m = N$ sharings in Σ.

- In Step 3, $\mathcal{S}$ emulates $\mathcal{F}_{\text{coin}}$ and generates a random field element $\alpha \in \mathbb{K}$. Then, $\mathcal{S}$ computes the shares of $[\![\boldsymbol{s}]\!]$ held by corrupted parties, i.e., $\pi_{\mathcal{C}orr}([\![\boldsymbol{s}]\!])$. Based on $\pi_{\mathcal{C}orr}([\![\boldsymbol{s}]\!])$, $\mathcal{S}$ randomly samples $[\![\boldsymbol{s}]\!] \in \Sigma^{\times m}(\mathcal{C}orr, \pi_{\mathcal{C}orr}([\![\boldsymbol{s}]\!]))$.
- In Step 4, $\mathcal{S}$ faithfully follows the protocol since the shares of $[\![\boldsymbol{s}]\!]$ held by honest parties have been explicitly generated.

When P_d is corrupted, $\mathcal{S}$ simply follows the protocol. If there exists some $[\![\boldsymbol{s}^{(\ell)}]\!]$ such that $\pi_{\mathcal{H}}([\![\boldsymbol{s}^{(\ell)}]\!])$ is invalid, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{rand-sharing}}$ and aborts in Step 4 (even if the verification passes). Otherwise, for each sharing $[\![\boldsymbol{s}^{(\ell)}]\!]$ dealt by P_d, $\mathcal{S}$ randomly samples $[\![\tilde{\boldsymbol{s}}^{(\ell)}]\!] \in \Sigma^{\times m}(\mathcal{H}, \pi_{\mathcal{H}}([\![\boldsymbol{s}^{(\ell)}]\!]))$, and views it as the sharing dealt by P_d.

If some party aborts at the end of VERSHARE, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{rand-sharing}}$ and aborts.

Simulation for CONVERT. Recall that CONVERT only involves local computation. For each sharing $[\![\boldsymbol{s}_d]\!]$, $\mathcal{S}$ has computed $\pi_{\mathcal{C}orr}([\![\boldsymbol{s}_d]\!])$ in the simulation of VERSHARE. In CONVERT, $\mathcal{S}$ computes $\pi_{\mathcal{C}orr}([\![\boldsymbol{r}_i]\!])$ and sends them to $\mathcal{F}_{\text{rand-sharing}}$.

Hybrid Arguments. Now, we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties with overwhelming probability. Consider the following hybrids.

Hybrid$_0$: The execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ simulates VERSHARE for honest parties when the dealer P_d is corrupted. Note that, $\mathcal{S}$ simply follows the protocol in this case and computes the shares held by corrupted parties. The only difference is that $\mathcal{S}$ will abort if there exists some $[\![\boldsymbol{s}^{(\ell)}]\!]$ such that $\pi_{\mathcal{H}}([\![\boldsymbol{s}^{(\ell)}]\!])$ is invalid even if the verification in Step 4 passes. According to Lemma 15.1, this happens with negligible probability. Therefore, the distribution of **Hybrid**$_1$ is statistically close to the distribution of **Hybrid**$_0$.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ first simulates VERSHARE for honest parties when the dealer P_d is honest. Then, for each $\ell \in \{1, \ldots, N'\}$, $\mathcal{S}$ re-samples a new random sharing $[\![\tilde{\boldsymbol{s}}^{(\ell)}]\!] \in \Sigma^{\times m}(\mathcal{C}orr, \pi_{\mathcal{C}orr}([\![\boldsymbol{s}^{(\ell)}]\!]))$. $\mathcal{S}$ takes $\{[\![\tilde{\boldsymbol{s}}^{(\ell)}]\!]\}_{\ell=1}^{N'}$ as the sharings dealt by P_d.

Note that in Step 1 and Step 2, $\mathcal{A}$ only receives $\pi_{\mathcal{C}orr}([\![\boldsymbol{s}^{(\ell)}]\!])$. Therefore, $[\![\boldsymbol{s}^{(\ell)}]\!]$ is a random sharing in $\Sigma^{\times m}(\mathcal{C}orr, \pi_{\mathcal{C}orr}([\![\boldsymbol{s}^{(\ell)}]\!]))$. In Step 3, after receiving $\alpha \in \mathbb{K}$ from $\mathcal{F}_{\text{coin}}$, all parties compute

$$[\![\boldsymbol{s}]\!] := [\![\boldsymbol{s}^{(0)}]\!] + \alpha \cdot [\![\boldsymbol{s}^{(1)}]\!] + \alpha^2 \cdot [\![\boldsymbol{s}^{(2)}]\!] + \ldots + \alpha^{N'} \cdot [\![\boldsymbol{s}^{(N')}]\!].$$

Therefore, $[\![\boldsymbol{s}]\!]$ is a random sharing in $\Sigma^{\times m}(\mathcal{C}orr, \pi_{\mathcal{C}orr}([\![\boldsymbol{s}]\!]))$. Note that $[\![\boldsymbol{s}]\!]$ is masked by a random sharing $[\![\boldsymbol{s}^{(0)}]\!]$. Thus, $[\![\boldsymbol{s}]\!]$ is independent of $\{[\![\boldsymbol{s}^{(\ell)}]\!]\}_{\ell=1}^{N'}$. Therefore, given the view of $\mathcal{A}$, each sharing $[\![\boldsymbol{s}^{(\ell)}]\!]$ is a random sharing in $\Sigma^{\times m}(\mathcal{C}orr, \pi_{\mathcal{C}orr}([\![\boldsymbol{s}^{(\ell)}]\!]))$. This means that $\mathcal{S}$ can re-sample and use a new random sharing $[\![\tilde{\boldsymbol{s}}^{(\ell)}]\!] \in \Sigma^{\times m}(\mathcal{C}orr, \pi_{\mathcal{C}orr}([\![\boldsymbol{s}^{(\ell)}]\!]))$ instead of using the sharing $[\![\boldsymbol{s}^{(\ell)}]\!]$ generated in the beginning.

Thus, the distribution of $\mathbf{Hybrid}_2$ is the same as the distribution of $\mathbf{Hybrid}_1$.

$\mathbf{Hybrid}_3$: In this hybrid, $\mathcal{S}$ does not re-sample the sharings $\{[\![\tilde{\boldsymbol{s}}^{(\ell)}]\!]\}_{\ell=1}^{N'}$. Instead, $\mathcal{S}$ simulates CONVERT for honest parties, which does not need to generate the whole sharing $[\![\tilde{\boldsymbol{s}}^{(\ell)}]\!]$.

Let $\boldsymbol{M}_{\mathcal{C}orr}$ denote the sub-matrix of $\boldsymbol{M}$ containing columns with indices in $\mathcal{C}orr$, and $\boldsymbol{M}_{\mathcal{H}}$ denote the sub-matrix containing columns with indices in $\mathcal{H}$. We have

$$([\![\boldsymbol{r}_1]\!], \ldots, [\![\boldsymbol{r}_{n-t}]\!])^{\mathrm{T}} = \boldsymbol{M} \cdot ([\![\boldsymbol{s}_1]\!], \ldots, [\![\boldsymbol{s}_n]\!])^{\mathrm{T}} = \boldsymbol{M}_{\mathcal{C}orr} \cdot ([\![\boldsymbol{s}_j]\!])_{j\in\mathcal{C}orr}^{\mathrm{T}} + \boldsymbol{M}_{\mathcal{H}} \cdot ([\![\boldsymbol{s}_j]\!])_{j\in\mathcal{H}}^{\mathrm{T}}.$$

Let

$$\begin{aligned} ([\![\boldsymbol{r}_1^{(\mathcal{H})}]\!], \ldots, [\![\boldsymbol{r}_{n-t}^{(\mathcal{H})}]\!])^{\mathrm{T}} &:= \boldsymbol{M}_{\mathcal{H}} \cdot ([\![\boldsymbol{s}_j]\!])_{j\in\mathcal{H}}^{\mathrm{T}} \\ ([\![\boldsymbol{r}_1^{(\mathcal{C}orr)}]\!], \ldots, [\![\boldsymbol{r}_{n-t}^{(\mathcal{C}orr)}]\!])^{\mathrm{T}} &:= \boldsymbol{M}_{\mathcal{C}orr} \cdot ([\![\boldsymbol{s}_j]\!])_{j\in\mathcal{C}orr}^{\mathrm{T}}. \end{aligned}$$

Then $([\![\boldsymbol{r}_1]\!], \ldots, [\![\boldsymbol{r}_{n-t}]\!]) = ([\![\boldsymbol{r}_1^{(\mathcal{C}orr)}]\!], \ldots, [\![\boldsymbol{r}_{n-t}^{(\mathcal{C}orr)}]\!]) + ([\![\boldsymbol{r}_1^{(\mathcal{H})}]\!], \ldots, [\![\boldsymbol{r}_{n-t}^{(\mathcal{H})}]\!])$.

Recall that $\boldsymbol{M}^{\mathrm{T}}$ is a Vandermonde matrix of size $n \times (n-t)$. Therefore $\boldsymbol{M}_{\mathcal{H}}^{\mathrm{T}}$ is a Vandermonde matrix of size $(n-t) \times (n-t)$, which is invertible. There is a one-to-one map from $([\![\boldsymbol{r}_1^{(\mathcal{H})}]\!], \ldots, [\![\boldsymbol{r}_{n-t}^{(\mathcal{H})}]\!])$ to $([\![\boldsymbol{s}_j]\!])_{j\in\mathcal{H}}$. Given $([\![\boldsymbol{s}_j]\!])_{j\in\mathcal{C}orr}$, there is a one-to-one map from $([\![\boldsymbol{r}_1]\!], \ldots, [\![\boldsymbol{r}_{n-t}]\!])$ to $([\![\boldsymbol{r}_1^{(\mathcal{H})}]\!], \ldots, [\![\boldsymbol{r}_{n-t}^{(\mathcal{H})}]\!])$. Recall that for each sharing $[\![\boldsymbol{s}_d]\!]$ dealt by a corrupted party P_d, $\mathcal{S}$ received the shares of honest parties $\pi_{\mathcal{H}}([\![\boldsymbol{s}_d]\!])$ and sampled a random sharing $[\![\tilde{\boldsymbol{s}}_d]\!] \in \Sigma^{\times m}(\mathcal{H}, \pi_{\mathcal{H}}([\![\boldsymbol{s}_d]\!]))$. Note that this may not be the sharing dealt by P_d since we do not know the shares held by corrupted parties. However, we show that this does not affect the distribution of the shares of honest parties generated by $\mathcal{F}_{\text{rand}}$.

To see this, note that for each valid $([\![\tilde{\boldsymbol{s}}_j]\!])_{j\in\mathcal{C}orr}$, $\mathcal{S}$ computes $\{\pi_{\mathcal{C}orr}([\![\boldsymbol{r}_j]\!])\}_{j=1}^{n-t}$ and sends them to $\mathcal{F}_{\text{rand-sharing}}$. Then for each $j \in \{1, \ldots, n-t\}$, $\mathcal{F}_{\text{rand-sharing}}$ samples a random sharing $[\![\tilde{\boldsymbol{r}}_j]\!] \in \Sigma^{\times m}(\mathcal{C}orr, \pi_{\mathcal{C}orr}([\![\boldsymbol{r}_j]\!]))$. These random sharings $\{[\![\tilde{\boldsymbol{r}}_j]\!]\}_{j=1}^{n-t}$ correspond to random sharings $([\![\tilde{\boldsymbol{s}}_j]\!])_{j\in\mathcal{H}}$, which are independent of $([\![\tilde{\boldsymbol{s}}_j]\!])_{j\in\mathcal{C}orr}$. Thus, the distribution of $\mathbf{Hybrid}_3$ is the same as the distribution of $\mathbf{Hybrid}_2$.

Note that $\mathbf{Hybrid}_3$ is the execution in the ideal world and $\mathbf{Hybrid}_3$ is statistically close to $\mathbf{Hybrid}_0$, the execution in the real world. □

Analysis of the Communication Complexity of BATCH-RAND(N). In Step 1, we need to invoke VERSHARE(P_d, N') for each party P_d. Therefore, the communication complexity of Step 1 is $O(N' \cdot n^2 \cdot m \cdot \ell + n^3 \cdot m)$ elements in $\mathbb{F}$. Step 2 and Step 3 do not require any communication. Recall that $N' = \frac{N}{m(n-t)}$ and $m = [\mathbb{K} : \mathbb{F}]$ is bounded by the security parameter κ. Therefore, the overall communication complexity of BATCH-RAND(N) is

$$O(N' \cdot n^2 \cdot m \cdot \ell + n^3 \cdot m) = O(N \cdot n^2/(n-t) \cdot \ell + n^3 \cdot \kappa)$$

elements in $\mathbb{F}$.

A Semi-Honest Version for $\mathcal{F}_{\text{rand-sharing}}$. We can obtain a semi-honest version for $\mathcal{F}_{\text{rand-sharing}}$ by only keeping the first step in VERSHARE(P_d, N'). In this way, the cost of preparing N random sharings in Σ is $O(N \cdot n^2/(n-t) \cdot \ell)$ elements in $\mathbb{F}$.

15.2 Preprocessing for Honest Majority with Malicious Security

We assume that the number of parties $n \equiv 1 \mod 4$. In the honest majority setting, the number of corrupted parties is $t = (n-1)/2$. Recall that in our protocol, the number k of secrets that can be packed within a single sharing should satisfy that $t = n - 2k + 1$. Therefore, $k = (n+3)/4$. Note that $t + k - 1 = n - k$.

Recall that in Chapter 11.2, a *degree-d* ($d \geq k-1$) packed Shamir sharing of $\boldsymbol{x} = (x_1, \ldots, x_k) \in \mathbb{F}^k$ (with the default positions $\boldsymbol{\beta}$) is a vector $(w_1, \ldots, w_n)$ for which there exists a polynomial $f(\cdot) \in \mathbb{F}[X]$ of degree at most d such that $\forall i \in \{1, 2, \ldots, k\}, f(\beta_i) = x_i$ and $\forall i \in \{1, 2, \ldots, n\}, f(\alpha_i) = w_i$, where $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_k$ are $n + k$ fixed and distinct elements in $\mathbb{F}$. For an element $x \in \mathbb{F}$, we define a degree-t Shamir sharing $[x|_j]_t$ to be a vector $(w_1, \ldots, w_n)$ for which there exists a polynomial $f(\cdot) \in F[X]$ of degree at most t such that $f(\beta_j) = x$ and $\forall i \in \{1, 2, \ldots, n\}, f(\alpha_i) = w_i$. In other words, $[x|_j]_t$ is a standard degree-t Shamir sharing except that the secret is hidden at the evaluation point β_j.

Let $\boldsymbol{e}_j$ be a vector in $\mathbb{F}^k$ where all entries are 0 except the j-th entry is 1. Let $[\boldsymbol{e}_j]_{k-1}$ denote the degree-$(k-1)$ packed Shamir sharing that is fully determined by the secret $\boldsymbol{e}_j$. Consider the multiplication $[\boldsymbol{e}_j]_{k-1} \cdot [x|_j]_t$. Let f denote the polynomial that corresponds to the resulting sharing. Since $[\boldsymbol{e}_j]_{k-1}$ corresponds to a degree-$(k-1)$ polynomial, and $[x|_j]_t$ corresponds to a degree-t polynomial, the polynomial f is of degree $t + k - 1 = n - k$. For all $j' \neq j$, since the j'-th entry of $\boldsymbol{e}_j$ is 0, the polynomial f satisfies that $f(\beta_{j'}) = 0$. As for j, the polynomial f satisfies that $f(\beta_j) = x$. Thus, the resulting sharing of $[\boldsymbol{e}_j]_{k-1} \cdot [x|_j]_t$ is a degree-$(n-k)$ packed Shamir sharing where the secret vector satisfies that all entries are 0 except the j-th entry is x.

In general, let $\boldsymbol{x} = (x_1, x_2, \ldots, x_k) \in \mathbb{F}^k$. All parties can locally transform k degree-t Shamir sharings $\{[x_j|_j]_t\}_{j=1}^k$ to a degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{x}]_{n-k}$ by computing

$$[\boldsymbol{x}]_{n-k} := \sum_{j=1}^{k} [\boldsymbol{e}_j]_{k-1} \cdot [x_j|_j]_t.$$

This reduces the problem of preparing random degree-$(n-k)$ packed Shamir sharings to preparing random degree-t Shamir sharings. In particular, we can use the DN multiplication protocol to prepare multiplication tuples for degree-t Shamir sharings.

Prepare Random Packed Shamir Sharings. We can directly use the protocol BATCH-RAND to prepare random degree-$(n-k)$ packed Shamir sharings and random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0} \in \mathbb{F}^k$. These random sharings are used in Protocol TRAN-MAL. Recall that $k = (n - t + 1)/2$. The amortized communication complexity per sharing is $\frac{n^2}{2k-1}$ elements.

Realizing $\mathcal{F}_{\text{prep-mal}}$. Our protocol will make use of the following two functionalities for multiplications between two degree-t Shamir sharings:

- The first functionality is $\mathcal{F}_{\text{mult-mal}}$ (See Chapter 7) which allows all parties to multiply two degree-t Shamir sharings. We restate the functionality below.
- Since $\mathcal{F}_{\text{mult-mal}}$ does not guarantee the correctness of the multiplication, all parties need to verify the the multiplications computed by $\mathcal{F}_{\text{mult-mal}}$. The second functionality $\mathcal{F}_{\text{multVerify}}$ (See Chapter 7) takes m multiplication tuples as input and outputs to all parties a single bit b indicating whether all multiplication tuples are correct. We restate the functionality below.

Recall that when using ATLAS-MULT (See Protocol 6.7 and Chapter 8.1) to instantiate $\mathcal{F}_{\text{mult-mal}}$, the communication complexity per multiplication gate is $4n$ elements. When using MULTVERIFICATION (See Protocol 7.10 and Chapter 7) to instantiate $\mathcal{F}_{\text{multVerify}}$, the communication complexity is $O(n^2 \cdot \kappa + n \cdot \kappa^2)$ elements, where κ is the security parameter.

Figure 7.1: Functionality $\mathcal{F}_{\text{mult-mal}}$

1. Let $[x]_t, [y]_t$ denote the input sharings. $\mathcal{F}_{\text{mult-mal}}$ receives from honest parties their shares of $[x]_t, [y]_t$. Then $\mathcal{F}_{\text{mult-mal}}$ reconstructs the secrets x, y. $\mathcal{F}_{\text{mult-mal}}$ further computes the shares of $[x]_t, [y]_t$ held by corrupted parties, and sends these shares to the adversary.
2. $\mathcal{F}_{\text{mult-mal}}$ receives from the adversary a value d and a set of shares $\{z_i\}_{i \in \mathcal{C}orr}$.
3. $\mathcal{F}_{\text{mult-mal}}$ computes $x \cdot y + d$. Based on the secret $z := x \cdot y + d$ and the t shares $\{z_i\}_{i \in \mathcal{C}orr}$, $\mathcal{F}_{\text{mult-mal}}$ reconstructs the whole sharing $[z]_t$ and distributes the shares of $[z]_t$ to honest parties.

For $\mathcal{F}_{\text{prep-mal}}$:

1. In Step 1, to prepare $[\boldsymbol{\gamma}]_{n-k}$, where $\boldsymbol{\gamma} = (\gamma, \gamma, \ldots, \gamma) \in \mathbb{F}^k$, all parties prepare random degree-t Shamir sharings $[\gamma|_1]_t, [\gamma|_2]_t, \ldots, [\gamma|_k]_t$ by using BATCH-RAND. For all $j \in \{1, 2, \ldots, k\}$, let $\boldsymbol{e}_j$ be a vector in $\mathbb{F}^k$ where all entries are 0 except the j-th entry is 1. Let $[\boldsymbol{e}_j]_{k-1}$ denote the degree-$(k-1)$ packed Shamir sharing that is fully determined by the secret $\boldsymbol{e}_j$. All parties compute

$$[\boldsymbol{\gamma}]_{n-k} = [\boldsymbol{e}_1]_{k-1} \cdot [\gamma|_1]_t + [\boldsymbol{e}_2]_{k-1} \cdot [\gamma|_2]_t + \ldots + [\boldsymbol{e}_k]_{k-1} \cdot [\gamma|_k]_t.$$

2. In Step 2, for every group of k input gates and output gates:
   (a) All parties prepare a random degree-$(n-k)$ packed Shamir sharing $[\boldsymbol{r}]_{n-k}$ by first preparing k random degree-t Shamir sharings $[r_1|_1]_t, [r_2|_2]_t, \ldots, [r_k|_k]_t$ by using BATCH-RAND and then locally computing

$$[\boldsymbol{r}]_{n-k} = [\boldsymbol{e}_1]_{k-1} \cdot [r_1|_1]_t + [\boldsymbol{e}_2]_{k-1} \cdot [r_2|_2]_t + \ldots + [\boldsymbol{e}_k]_{k-1} \cdot [r_k|_k]_t.$$

Figure 7.2: Functionality $\mathcal{F}_{\text{multVerify}}$

1. Let N denote the number of multiplication tuples. The multiplication tuples are denoted by

$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(N)}]_t.[y^{(N)}]_t, [z^{(N)}]_t).$$

2. For all $i \in [N]$, $\mathcal{F}_{\text{multVerify}}$ receives from honest parties their shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$. Then $\mathcal{F}_{\text{multVerify}}$ reconstructs the secrets $x^{(i)}, y^{(i)}, z^{(i)}$. $\mathcal{F}_{\text{multVerify}}$ further computes the shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$ held by corrupted parties and sends these shares to the adversary.
3. For all $i \in [N]$, $\mathcal{F}_{\text{multVerify}}$ computes $d^{(i)} = z^{(i)} - x^{(i)} \cdot y^{(i)}$ and sends $d^{(i)}$ to the adversary.
4. Finally, let $b \in \{\texttt{abort}, \texttt{accept}\}$ denote whether there exists $i \in [N]$ such that $d^{(i)} \neq 0$. $\mathcal{F}_{\text{multVerify}}$ sends b to the adversary and waits for its response.
   - If the adversary replies `continue`, $\mathcal{F}_{\text{multVerify}}$ sends b to honest parties.
   - If the adversary replies `abort`, $\mathcal{F}_{\text{multVerify}}$ sends `abort` to honest parties.

For all $j \in \{1, 2, \ldots, k\}$, all parties invoke $\mathcal{F}_{\text{mult-mal}}$ on $[r_j|_j]_t, [\gamma|_j]_t$ and obtain $[\gamma \cdot r_j|_j]_t$. Finally, all parties compute $[\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k}$ by

$$[\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k} = \sum_{j=1}^{k} [\boldsymbol{e}_j]_{k-1} \cdot [\gamma \cdot r_j|_j]_t.$$

(b) All parties prepare $[\boldsymbol{\beta}]_{n-k}, [\boldsymbol{\beta} * \boldsymbol{r}]_{n-k}$ in the same way as that for $[\boldsymbol{r}]_{n-k}, [\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k}$ in Step 2.(a).

(c) For every group of k output gates, all parties prepare a random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0}$ by using BATCH-RAND.

3. In Step 3, for every group of k multiplication gates:

(a) All parties first prepare $[\![\boldsymbol{a}]\!]_{n-k}, [\![\boldsymbol{b}]\!]_{n-k}$ in the same way as that for $[\![\boldsymbol{r}]\!]_{n-k}$ in Step 2.(a). Note that all parties also hold $\{[a_j|_j]_t\}_{j=1}^k$ and $\{[b_j|_j]_t\}_{j=1}^k$. Then for all $j \in \{1, 2, \ldots, k\}$, all parties invoke $\mathcal{F}_{\text{mult-mal}}$ on $[a_j|_j]_t, [b_j|_j]_t$ and obtain $[a_j \cdot b_j|_j]_t$. Next, all parties compute $[\boldsymbol{a} * \boldsymbol{b}]_{n-k}$ by

$$[\boldsymbol{a} * \boldsymbol{b}]_{n-k} = \sum_{j=1}^{k} [\boldsymbol{e}_j]_{k-1} \cdot [a_j \cdot b_j|_j]_t.$$

Finally, all parties compute $[\boldsymbol{\gamma} * \boldsymbol{c}]_{n-k}$ in the same way as that for $[\boldsymbol{\gamma} * \boldsymbol{r}]_{n-k}$ in Step 2.(a).

(b) All parties prepare three two degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$ by using BATCH-RAND.

4. In Step 4, all parties prepare three two degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$ by using BATCH-RAND.
5. Finally, all parties verify the correctness of the multiplications by invoking $\mathcal{F}_{\text{multVerify}}$. For all $j \in \{1, 2, \ldots, k\}$, all multiplication tuples where the secrets are hidden at the evaluation point β_j are verified together.

We note that, with honest majority, the functionality $\mathcal{F}_{\text{com}}$ can be realized without any preprocessing data. The committer simply generates a random degree-t Shamir sharing of the value he wants to commit and distributes the shares to other parties. To open the commitment, all parties exchange their shares, check whether the shares form a valid degree-t Shamir sharing, and reconstruct the secret if the check passes. The secrecy follows from the following two facts:

- The shares of a degree-t Shamir sharing of honest parties fully determine the secret. Therefore, the committer or corrupted parties cannot change the secret after the sharing is distributed.
- The shares of a degree-t Shamir sharing of corrupted parties are independent of the secret. Therefore, from the shares they received, corrupted parties cannot learn any information about the value committed by an honest party.

Theorem 15.1. *In the client-server model, let* c *denote the number of clients, and* $n = 2t + 1$ *denote the number of parties (servers). Let* κ *be the security parameter and* $\mathbb{F}$ *be a finite field of size* $|\mathbb{F}| \geq 2^{\kappa}$*. For an arithmetic circuit* C *over* $\mathbb{F}$*, there exists an information-theoretic MPC protocol which securely computes the arithmetic circuit* C *in the presence of a fully malicious adversary controlling up to* c *clients and* t *parties. The cost of the protocol is* $O(|C| \cdot n + \textit{poly}(\mathsf{Depth}, \mathsf{c}, n))$ *elements of offline communication and* $O(|C| + \textit{poly}(\mathsf{Depth}, \mathsf{c}, n))$ *elements of online communication, where* Depth *is the circuit depth.*

Bibliography

[1] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient Information-Theoretic Secure Multiparty Computation over $\mathbb{Z}/p^k\mathbb{Z}$ via Galois Rings. In *Theory of Cryptography*, pages 471–501, Cham, 2019. Springer International Publishing. ISBN 978-3-030-36030-6. 11.4

[2] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, Matthieu Rambaud, Chaoping Xing, and Chen Yuan. Asymptotically Good Multiplicative LSSS over Galois Rings and Applications to MPC over $\mathbb{Z}/p^k\mathbb{Z}$. In *Advances in Cryptology – ASIACRYPT 2020*, pages 151–180, Cham, 2020. Springer International Publishing. ISBN 978-3-030-64840-4. 11.1

[3] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. Cryptology ePrint Archive, Paper 2021/576, 2021. URL `https://eprint.iacr.org/2021/576`. `https://eprint.iacr.org/2021/576`. 9.2

[4] 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 adversariesłbreaking the 1 billion-gate per second barrier. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 843–862. IEEE, 2017. 3, 3.2

[5] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991. 2b, 10.2.2

[6] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-46766-3. 4.3

[7] Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-c secure multiparty computation for highly repetitive circuits. In *Advances in Cryptology – EUROCRYPT 2021*, pages 663–693, Cham, 2021. Springer International Publishing. ISBN 978-3-030-77886-6. 1.3, 1.3, 9, 9, 9.2, 10

[8] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In *Theory of Cryptography*, pages 213–230, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78524-8. 3.2

[9] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10. ACM, 1988. 1, 1.2, 1.2, 1.3,

3.2, 9, 10

[10] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *Advances in Cryptology – CRYPTO 2012*, pages 663–680, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32009-5. 1.2, 3, 3.2, 3.2, 4.4, 4.4.1, 4.4.3, 7.2, 7.2, 15.1, 15.1.2

[11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 169–188, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-20465-4. 10.2.5, 14

[12] Fabrice Benhamouda, Elette Boyle, Niv Gilboa, Shai Halevi, Yuval Ishai, and Ariel Nof. Generalized pseudorandom secret sharing and efficient straggler-resilient secure computation. In Kobbi Nissim and Brent Waters, editors, *Theory of Cryptography*, pages 129–161, Cham, 2021. Springer International Publishing. ISBN 978-3-030-90453-1. 9.2, 9.2

[13] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *Advances in Cryptology – CRYPTO 2019*, pages 67–97, Cham, 2019. Springer International Publishing. ISBN 978-3-030-26954-8. 1.2, 3, 3.1, 3.2, 4.4.3, 1, 9

[14] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 489–518, Cham, 2019. Springer International Publishing. ISBN 978-3-030-26954-8. 9.2

[15] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. Cryptology ePrint Archive, Paper 2020/1392, 2020. URL `https://eprint.iacr.org/2020/1392`. `https://eprint.iacr.org/2020/1392`. 9.2

[16] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficientpseudorandomcorrelationgenerators from ring-lpn. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 387–416, Cham, 2020. Springer International Publishing. ISBN 978-3-030-56880-1. 9.2

[17] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Efficient fully secure computation via distributed zero-knowledge proofs. In *Advances in Cryptology – ASIACRYPT 2020*, pages 244–276, Cham, 2020. Springer International Publishing. ISBN 978-3-030-64840-4. 1.2, 1.2, 3, 3.2, 1, 9, 9

[18] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Sublinear GMW-Style Compiler for MPC with Preprocessing. In *Advances in Cryptology – CRYPTO 2021*, pages 457–485, Cham, 2021. Springer International Publishing. ISBN 978-3-030-84245-1. 2.2, 10.2

[19] Gabriel Bracha. An o(log n) expected rounds randomized byzantine generals protocol. *J. ACM*, 34(4):910920, oct 1987. ISSN 0004-5411. doi: 10.1145/31846.42229. URL `https://doi.org/10.1145/31846.42229`. 9.2

[20] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000. doi: 10.1007/s001459910006. URL https://doi.org/10.1007/s001459910006. 2.2

[21] Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized Complexity of Information-Theoretically Secure MPC Revisited. In *Advances in Cryptology – CRYPTO 2018*, pages 395–426, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96878-0. 9, 10, 11.4, 15.1, 15.1, 15.1

[22] David Chaum, Ivan B Damgård, and Jeroen Van de Graaf. Multiparty computations ensuring privacy of each partys input and correctness of the result. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 87–119. Springer, 1987. 3.2

[23] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19. ACM, 1988. 1.2, 3.2

[24] 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 *Annual International Cryptology Conference*, pages 34–64. Springer, 2018. 1.2, 1.2, 1.2, 3, 3.1, 3.2, 4.4, 4.4.2, 4.4.2, 4.1, 4.4.3, 7, 7.1, 9, 9

[25] Geoffroy Couteau. A Note on the Communication Complexity of Multiparty Computation in the Correlated Randomness Model. In *Advances in Cryptology – EUROCRYPT 2019*, pages 473–503, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17656-3. 2.2, 9.2, 10.2

[26] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *Theory of Cryptography*, pages 342–362, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30576-7. 9.2

[27] Ronald Cramer, Matthieu Rambaud, and Chaoping Xing. Asymptotically-Good Arithmetic Secret Sharing over $\mathbb{Z}/p^{\ell}\mathbb{Z}$ with Strong Multiplication and Its Applications to Efficient MPC. In *Advances in Cryptology – CRYPTO 2021*, pages 656–686, Cham, 2021. Springer International Publishing. ISBN 978-3-030-84252-9. 9, 10, 11.4

[28] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*, pages 572–590. Springer, 2007. (document), 1.2, 1.2, 3, 3.2, 3.2, 4.1, 2, 4.1, 4.2, 4.2, 4.3, 4.3, 4.3, 4.4, 4.4.1, 4.4.3, 5.2, 5.3, 6, 6.1, 6.1, 6.1, 6.1, 6.1, 6.1, 6.2, 6.2.2, 6.2.3, 6.3, 6.3.1, 6.3.1, 6.3.2, 6.3.2, 6.3.2, 6.3.4, 7.1, 7.4, 7.4, 9, 9, 2, 10

[29] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 445–465. Springer, 2010. 1.3, 1.3, 9, 9, 9.2, 9.2, 9.2, 10

[30] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012*, pages 643–662. Springer, 2012. 1.3, 1.3, 9, 9.2, 9.2, 10.2.5, 14, 14.3.4

[31] Ivan Damgård, Jesper Buus Nielsen, Antigoni Polychroniadou, and Michael Raskin. On the communication required for unconditionally secure multiplication. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 459–488, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53008-5. 4.3

[32] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - a framework for efficient mixed-protocol secure two-party computation. NDSS, 2015. 9.2

[33] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 823–852, Cham, 2020. Springer International Publishing. ISBN 978-3-030-56880-1. 9.2

[34] Matthew Franklin and Moti Yung. Communication Complexity of Secure Computation (Extended Abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, page 699710, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897915119. doi: 10.1145/129712.129780. URL https://doi.org/10.1145/129712.129780. 1.3, 1.3, 9, 9, 10.1.2, 10.1.2, 11.2

[35] Jun Furukawa and Yehuda Lindell. Two-thirds honest-majority mpc for malicious adversaries at almost the cost of semi-honest. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1557–1571, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3339811. URL https://doi.org/10.1145/3319535.3339811. 3

[36] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 225–255. Springer, 2017. 3.2

[37] Juan Garay, Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. The price of low communication in secure multi-party computation. In *Advances in Cryptology – CRYPTO 2017*, pages 420–446, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63688-7. 9

[38] Daniel Genkin, Yuval Ishai, Manoj M. Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 495–504, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2710-7. doi: 10.1145/2591796.2591861. URL http://doi.acm.org/10.1145/2591796.2591861. 1.2, 1.2, 3, 3.2, 4.4, 7, 7.1, 7.1, 9

[39] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: from passive to active security via secure simd circuits. In *Annual Cryptology Conference*, pages 721–741. Springer, 2015. 1.3, 1.3, 9, 9, 9.2, 10

[40] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987. 1, 3, 3.2

[41] S. Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The More the Merrier: Reducing the Cost of Large Scale MPC. In *Advances in Cryptology – EUROCRYPT 2021*, pages 694–723, Cham, 2021. Springer International Publishing. ISBN 978-3-030-77886-6. 1.3, 1.3, 9, 9, 9.2

[42] Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority mpc. Cryptology ePrint Archive, Report 2020/134, 2020. https://eprint.iacr.org/2020/134. 6.3.2

[43] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional mpc with guaranteed output delivery. In *Advances in Cryptology – CRYPTO 2019*, pages 85–114, Cham, 2019. Springer International Publishing. ISBN 978-3-030-26951-7. 3.2

[44] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority mpc. In *Advances in Cryptology – CRYPTO 2020*, pages 618–646, Cham, 2020. Springer International Publishing. ISBN 978-3-030-56880-1. 1.2, 1.2, 3, 3.2, 6.2, 6.2.2, 9

[45] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. Atlas: Efficient and scalable mpc inthe honest majority setting. In *Advances in Cryptology – CRYPTO 2021*, pages 244–274, Cham, 2021. Springer International Publishing. ISBN 978-3-030-84245-1. 1.2, 3.2, 9

[46] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional Communication-Efficient MPC via Hall's Marriage Theorem. In *Advances in Cryptology – CRYPTO 2021*, pages 275–304, Cham, 2021. Springer International Publishing. ISBN 978-3-030-84245-1. 1.3, 9.2

[47] Brett Hemenway, Steve Lu, Rafail Ostrovsky, and William Welser IV. High-precision secure computation of satellite collision probabilities. In Vassilis Zikas and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 169–187, Cham, 2016. Springer International Publishing. ISBN 978-3-319-44618-9. 9.2

[48] Martin Hirt and Ueli Maurer. Robustness for free in unconditional multi-party computation. In *Annual International Cryptology Conference*, pages 101–118. Springer, 2001. 3.2

[49] Martin Hirt, Ueli Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 143–161. Springer, 2000. 3.2

[50] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591. Springer, 2008. doi: 10.1007/978-3-540-85174-5_32. URL https://doi.org/10.1007/978-3-540-85174-5_32. 1.3

[51] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In *Proceedings of the 10th Theory of Cryptography Conference on Theory of Cryptography*, TCC'13, page 600620, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 9783642365935. doi: 10.1007/978-3-642-36594-2_34. URL https://doi.org/10.

1007/978-3-642-36594-2_34. 9.2

[52] Yuval Ishai, Eyal Kushilevitz, Manoj Prabhakaran, Amit Sahai, and Ching-Hua Yu. Secure protocol transformations. In *Advances in Cryptology – CRYPTO 2016*, pages 430–458, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53008-5. 1.2, 9.2

[53] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 830842, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978357. URL https://doi.org/10.1145/2976749.2978357. 9.2

[54] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making spdz great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 158–189, Cham, 2018. Springer International Publishing. ISBN 978-3-319-78372-7. 9.2

[55] Joe Kilian. Founding crytpography on oblivious transfer. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 2031, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912640. doi: 10.1145/62212.62215. URL https://doi.org/10.1145/62212.62215. 1.3

[56] Yehuda Lindell and Ariel Nof. A framework for constructing fast mpc over arithmetic circuits with malicious adversaries and an honest-majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 259–276. ACM, 2017. 1.2, 3.2

[57] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of cryptology*, 25(4):680–722, 2012. 3.2

[58] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 3552, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243760. URL https://doi.org/10.1145/3243734.3243760. 9.2

[59] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology–CRYPTO 2012*, pages 681–700. Springer, 2012. 3.2

[60] Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority mpc by batchwise multiplication verification. In *Applied Cryptography and Network Security*, pages 321–339, Cham, 2018. Springer International Publishing. ISBN 978-3-319-93387-0. 1.2, 1.2, 3, 3.1, 3.2, 4.4, 4.4.1, 4.4.2, 4.4.2, 4.4.3, 4.1, 4.4.3, 7.2, 7.2, 9

[61] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved Mixed-Protocol secure Two-Party computation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2165–2182. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/

usenixsecurity21/presentation/patra. 9.2

[62] Antigoni Polychroniadou and Yifan Song. Constant-Overhead Unconditionally Secure Multiparty Computation Over Binary Fields. In *Advances in Cryptology – EUROCRYPT 2021*, pages 812–841, Cham, 2021. Springer International Publishing. ISBN 978-3-030-77886-6. 9, 10, 11.4, 15, 15.1

[63] Dragos Rotaru and Tim Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology – INDOCRYPT 2019*, pages 227–249, Cham, 2019. Springer International Publishing. ISBN 978-3-030-35423-7. 9.2

[64] Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, November 1979. ISSN 0001-0782. doi: 10.1145/359168.359176. URL http://doi.acm.org/10.1145/359168.359176. 3, 4, 5.1, 10.1.2, 11.2

[65] Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982. 1, 3, 3.2