Unconditional Communication-Efficient MPC via Hall's Marriage Theorem

Vipul Goyal
Carnegie Mellon University
and NTT Research
goyal@cs.cmu.edu

Antigoni Polychroniadou
J.P. Morgan AI Research
antigonipoly@gmail.com

Yifan Song
Carnegie Mellon University
yifans2@andrew.cmu.edu

Abstract

The best known n party unconditional multiparty computation protocols with an optimal corruption threshold communicates $O(n)$ field elements per gate. This has been the case even in the semi-honest setting despite over a decade of research on communication complexity in this setting. Going to the slightly sub-optimal corruption setting, the work of Damgård, Ishai, and Krøigaard (EUROCRYPT 2010) provided the first protocol for a single circuit achieving communication complexity of $O(\log |C|)$ elements per gate. While a number of works have improved upon this result, obtaining a protocol with $O(1)$ field elements per gate has been an open problem.

In this work, we construct the first unconditional multi-party computation protocol evaluating a single arithmetic circuit with amortized communication complexity of $O(1)$ elements per gate.

1 Introduction

Secure Multi-Party Computation (MPC) enables a set of n parties to mutually run a protocol that computes some function f on their private inputs without compromising the privacy of their inputs or the correctness of the outputs [Yao82, GMW87, CCD88, BOGW88]. An important distinction in designing MPC protocols is that of the power of the adversary. An adversary in a semi-honest protocol follows the protocol's specification but tries to learn information from the received messages, and an adversary in a malicious protocol is allowed to deviate from the protocol's specification in arbitrary ways.

In this work, our focus is on the communication complexity of information theoretic protocols evaluating an arithmetic circuit in the presence of semi-honest or malicious adversaries. The "dream" in the unconditional setting is to get as close to $|C|$ as possible (or even below) where $|C|$ is the circuit size. The best known protocols in the so called optimal threshold regime tolerating $t = (n-1)/2$ corrupted parties require communicating $O(n \cdot |C|)$ field elements (ignoring circuit independent terms) [DN07, GIP$^+$14, CGH$^+$18,

V. Goyal, Y. Song—Supported in part by the NSF award 1916939, DARPA SIEVE program, a gift from Ripple, a DoE NETL award, a JP Morgan Faculty Fellowship, a PNC center for financial services innovation award, and a Cylab seed funding award.

NV18, BBCG+19, GSZ20, BGIN20]. There are no constructions known beating this barrier even in the semi-honest setting despite over a decade of research.

Moving to Sub-optimal Corruption Threshold. In a remarkable result, Damgård et al. [DIK10] showed an unconditional MPC protocol with communication complexity of $O(\log |C| \cdot n/k)$ per gate (ignoring circuit independent terms) tolerating $t' = (n-1)/3 - k + 1$ corrupted parties. This was later extended by Genkin et al. [GIP15] to obtain a construction tolerating $t' = (n-1)/2 - k + 1$ 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 [FY92] where k secrets are packed into a single secret sharing. An incomparable result was given by Garay et al. [GIOZ17] who obtained a protocol with communication complexity $O(\log^{1+\delta} n \cdot |C|)$ where δ is any positive constant. If one was interested in evaluating the same circuit multiple times on different inputs, *Franklin and Yung* [FY92] showed how to use packed secret sharing to evaluate k copies of the circuit with *amortized* communication complexity of $O(n/k)$ elements per gate or $O(1)$ elements when $k = O(n)$. However in case of a single circuit evaluation, the works mentioned [GIP15, GIOZ17] remains the best known.

To our knowledge, there is no known unconditional MPC protocol which only requires communicating $O(1)$ field elements per gate for any corruption threshold (assuming the number of corrupted parties is at least super-constant). This raises the following natural question:

Is it possible to construct information theoretic MPC protocols for computing a single *arithmetic circuit with communication complexity $O(1)$ field elements per gate?*

We answer the above question in the affirmative by constructing an information theoretic n-party protocol based on packed secret sharing for an arithmetic circuit over a finite field $\mathbb{F}$ of size $|\mathbb{F}| \geq 2n$. Our communication complexity amortized over the multiplication gates within the same circuit (rather than amortized over multiple circuits) is $O(n/k)$ field elements per multiplication gate. Informally, we prove the following:

Theorem 1 (informal). *Assume a point-to-point channel between every pair or parties. For all $1 \leq k \leq t$ where $t = \lfloor (n-1)/2 \rfloor$, there exists an information theoretic n-party MPC protocol which securely computes a single arithmetic circuit in the presence of a semi-honest (malicious) adversary controlling up to $t - k + 1$ parties with an communication complexity of $O(n/k)$ field elements per multiplication gate. For the case where $k = O(n)$, the achieved communication complexity is $O(1)$ elements per gate. In addition, our finite field $\mathbb{F}$ is of size $|\mathbb{F}| \geq 2n$.*

Our formal theorem for semi-honest security (with perfect security) can be found in Theorem 7 and for malicious security (with abort and statistical security) in Theorem 8. In order to achieve these results, we introduce a set of combinatorial lemmas which could be of independent interest. In particular, we marry packed secret sharing with techniques from graph theory. A key technical challenge with using packed secret sharing in the context of a single circuit is to make sure that all the required secrets for a batch of gates appear in a single packed secret sharing. In addition, one needs to ensure that these secrets appear in the correct order. Our key technical contributions in this paper relate to performing secure permutations of the secrets efficiency by using techniques from perfect matching in bipartite graphs. In particular, we make an extensive use of Hall's Marriage Theorem.

1.1 Related Works

We note that the work [GIOZ17] focuses on the same setting as our paper (i.e., the corruption threshold is $t' < (1/2 - \epsilon)n$ but uses a different approach to evaluate a single circuit. Their goal is to achieve communication complexity that is sublinear in the total number of parties. The high-level idea is to first select a small subset of parties as the committee, which follows honest majority with overwhelming probability, and then evaluate the circuit using an efficient protocol in the honest majority setting among the committee. As a result, their protocol achieves communication complexity of $O(\log^{1+\delta} n \cdot |C|)$ elements with error probability negligible in n for any constant $\delta > 0$. Compared with ours, when measuring the communication complexity per gate, our protocol achieves $O(1)$ elements while the protocol in [GIOZ17] requires $O(\log^{1+\delta} n)$ elements. In Appendix C, we show that we can combine the techniques in [GIOZ17] with ours to further reduce the communication complexity.

Recently, two independent works [GSY21, BGJK21] also use the packed secret sharing technique in the MPC paradigm. The work [GSY21] uses the packed secret sharing to prepare Beaver triples in the preparation phase. Then these Beaver triples are unpacked by using additive sharings and distributed to the parties for the online phase. The use of packed secret sharing allows Gorden, et al. [GSY21] to achieve $O(1)$ elements per Beaver triple in the communication complexity during the preparation phase. However, these Beaver triples are used *individually* after unpacking, which leads to $O(n)$ elements per gate in the online phase. Gorden, et al. [GSY21] discusses how to use (one or more) small committees to mitigate the communication cost in the online phase. But they still do not achieve $O(1)$ elements per gate in the online phase. Moreover, the work [GSY21] highly relies on computational assumptions.

The work [BGJK21] studies a special class of circuits that have a highly repetitive structure. Informally, a highly repetitive circuit is composed of several blocks with sufficient width that recur sufficient number of times throughout the circuit. Beck, et al. [BGJK21] show that $O(1)$ elements per gate can be achieved for such a kind of circuits. Technically, the highly repetitive structure allows Beck, et al. [BGJK21] to efficiently prepare random sharings to perform permutations on the secrets in a single sharing. Informally, when using the packed secret sharing technique to evaluate a batch of (multiplication or addition) gates, we need to ensure that the secrets of the two input sharings are in the correct order, i.e., for each input sharing, the i-th secret corresponds to the input of the i-th gate. To achieve the correct order, we need to permute the secrets in a single sharing. Known techniques from [DIK10] require all parties to first prepare a pair of random sharings in a specific structure that is related to the permutation we want to perform. The main bottleneck is that such a pair of random sharings can only be efficiently prepared in a batch way (of batch size $O(n)$). It requires that this permutation needs to be performed at least $O(n)$ times to achieve the communication complexity $O(1)$ elements per gate. See more discussions about this overhead in Section 2.2. The highly repetitive structure ensures this property directly since each kind of permutations that is required to perform in the circuit is guaranteed to be used at least once in each recursion. Compared with [BGJK21], we achieve the same communication complexity, i.e., $O(1)$ elements per gate, with no requirement in the circuit structure.

The notion of MPC was first introduced in [Yao82, GMW87] in 1980s. Feasibility results for MPC were obtained by [Yao82, GMW87] under cryptographic assumptions, and by [BOGW88, CCD88] in the information-theoretic setting. Subsequently, a large number of works have focused on improving the efficiency of MPC protocols in various settings.

In the setting of honest majority, a rich line of works focus on malicious security-with-abort and improving the communication efficiency [DN07, GIP$^+$14, CGH$^+$18, NV18, BBCG$^+$19, GSZ20, BGIN20]. When assuming the existence of a broadcast channel, the works [BSFO12, GSZ20] have shown that guaranteed output delivery can also be achieved efficiently.

It is known that perfect security (where there is no error and the protocol is guaranteed to succeed) requires the corruption threshold $t < n/3$. In this setting, a series of works [HMP00, HM01, DN07, BTH08, GLS19] focused on improving the asymptotic communication complexity.

2 Technical Overview

In the following, we will use $n = 2t + 1$ to denote the number of parties. Let $1 \leq k \leq t$ be an integer. We consider the scenario where an adversary is allowed to corrupt $t' = t - k + 1$ parties. For simplicity, we focus on the semi-honest setting. We will discuss how to achieve malicious security at a later point.

Our construction will use the packed secret-sharing technique introduced by Franklin and Yung [FY92]. This is a generalization of the standard Shamir secret sharing scheme [Sha79]. It allows to secret-share a batch of secrets within a single Shamir sharing. In the case that $t' = t - k + 1$, we can use a degree-t Shamir sharing, which requires $t+1$ shares to reconstruct the whole sharing, to store k secrets such that any t' shares are independent of the secrets. We refer to such a sharing as a degree-t packed Shamir sharing. Let $\boldsymbol{x}$ be a vector of dimension k. We use $[\boldsymbol{x}]$ to denote a degree-t packed Shamir sharing of the secrets $\boldsymbol{x}$.

In this work, we are interested in the information-theoretic setting. Our goal is to construct a semi-honest MPC protocol for a *single* arithmetic circuit over a finite field $\mathbb{F}$ (of size $|\mathbb{F}| \geq 2n$), such that the amortized communication complexity (of each party) per gate is $O(n/k)$ elements. Note that when $k = O(n)$, the amortized communication complexity per gate becomes $O(1)$ elements.

2.1 Background: Using the Packed Secret-sharing Technique in MPC

In the information-theoretic setting, a general approach to construct an MPC protocol is to compute a secret sharing for each wire of the circuit. The circuit is evaluated gate by gate, and the problem is reduced to compute the output sharing of an addition gate or a multiplication gate given the input sharings. When the corruption threshold can be relaxed to $t' = t - k + 1$ where $t = \frac{n-1}{2}$, a natural way of using the packed secret-sharing technique [FY92] is to compute $k \geq 1$ copies of the same circuit (i.e., a SIMD circuit): by storing the value related to the i-th copy in the i-th position of the secret sharing for each wire, all copies of the same circuit are evaluated simultaneously. Moreover, the communication complexity of a single operation for packed secret sharings is usually the same as that for standard secret sharings. Effectively, the amortized communication complexity per copy is reduced by a factor of k.

In 2010, Damgård et al. [DIK10] provided the first protocol of using packed secret-sharing technique to evaluate *a single circuit*. The original work focuses on the corruption threshold $t' < (1/3 - \epsilon)n$ and perfect security. It is later extended by [GIP15] to the setting of security with abort against $t' < (1/2-\epsilon)n$ corrupted parties with a constant factor improvement in the communication complexity[1]. At a high-level, the idea is to divide the gates of the same type in each layer into groups of k. Each group of gates will be evaluated at the same time. For each group of gates, all parties need to prepare the input sharings by using the output sharings from previous layers. Unlike the case when evaluating a SIMD circuit, input sharings for each group of gates do not come for free:

- The secrets needed to be in a single sharing may be scattered in different output sharings of previous layers.
- Even if we have all the secrets in a single sharing, we need the secrets to be in the correct order so that the i-th secret is the input of the i-th gate.

The naive approach of preparing a single input sharing by collecting the secret one by one would require $O(k)$ operations, which eliminates the benefit of using the packed secret-sharing technique. In [DIK10], they solve this problem by compiling the circuit into a special form of a universal circuit such that it can be viewed as k copies of the same circuit. In particular, the compilation uses the so-called Beneš network, which increases the circuit size by a factor of $\log |C|$, where $|C|$ is the circuit size. As a result, the amortized communication complexity per gate is $O(\log |C| \cdot n/k)$ elements.

Our work aims to remove the $\log |C|$ factor in the communication complexity and achieves the same communication efficiency as that for the evaluation of many copies of the same circuit. In this paper, we describe our idea from the bottom up:

1. We start with the basic protocols to evaluate input gates, addition gates, multiplication gates, and output gates using the packed Shamir sharing scheme. These protocols are simple variants of the protocols in [DN07], which focuses on the adversary that can corrupt t parties.
2. To use these protocols to evaluate addition gates and multiplication gates, we need the secrets in the input packed Shamir sharings to have the correct order. Assuming each input sharing contains all the secrets we want, we discuss how to permute the secrets in each input sharing to the correct order.

[1]While the semi-honest version of the protocol in [GIP15] can use a field $\mathbb{F}$ of size $O(n)$, the maliciously secure protocol requires to use a large enough field since the error probability is proportional to the field size.

3. Next, we show how to collect the secrets of an input packed Shamir sharing from the output sharings of previous layers. Our solution requires that each output wire from each layer is only used once in the computation, as an input wire to a single layer. This requirement can be met by further requiring that there is a fan-out gate right after each gate that copies the output wire the number of times it is used in later layers.

4. After that, we discuss how to evaluate fan-out gates efficiently.

5. Finally, we discuss how to achieve malicious security.

Our key techniques lie on the second point and the third point. We will focus on these two points in the technical overview, which are in Section 2.2 and Section 2.3. We will briefly discuss the last two points in Section 2.4 and Section 2.6.

2.2 Performing an Arbitrary Permutation on the Secrets of a Single Sharing

During the computation, we may encounter the scenario that the order of the secrets is not what we want. For example, when $k = 2$ and we want to compute two multiplication gates with input secrets $(x_1, y_1), (x_2, y_2)$, ideally we want all parties to hold two packed Shamir sharings of $\boldsymbol{x} = (x_1, x_2)$ and $\boldsymbol{y} = (y_1, y_2)$ so that when we use the multiplication protocol with these two packed Shamir sharing, we can obtain a packed Shamir sharing of the secret $\boldsymbol{x} * \boldsymbol{y} = (x_1 \cdot y_1, x_2 \cdot y_2)$. During the computation, however, all parties may hold two packed Shamir sharings of $\boldsymbol{x} = (x_1, x_2)$ and $\boldsymbol{y'} = (y_2, y_1)$. In particular, the secrets in the second sharing are not in the order we want. Using these two packed Shamir sharings in the multiplication protocol, we can only obtain a packed Shamir sharing of $\boldsymbol{x} * \boldsymbol{y'} = (x_1 \cdot y_2, x_2 \cdot y_1)$ instead of the correct result $\boldsymbol{x} * \boldsymbol{y} = (x_1 \cdot y_1, x_2 \cdot y_2)$.

To solve it, we need to construct a protocol which allows all parties to perform an arbitrary permutation on the secrets of a single sharing. Let $p(\cdot)$ be a permutation over $\{1, 2, \ldots, k\}$. We use F_p to denote the linear map which maps $\boldsymbol{x} = (x_1, x_2, \ldots, x_k)$ to $\tilde{\boldsymbol{x}} = (x_{p(1)}, x_{p(2)}, \ldots, x_{p(k)})$. Given the input sharing $[\boldsymbol{x}]$, the goal is to compute a degree-t packed Shamir sharing $[F_p(\boldsymbol{x})]$.

We first review the approach in [DIK10] for permuting the secrets of $[\boldsymbol{x}]$:

1. All parties prepare two random degree-t packed Shamir sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$, where $\tilde{\boldsymbol{r}} = F_p(\boldsymbol{r})$ and $p(\cdot)$ is the permutation we want to perform.

2. All parties locally compute $[\boldsymbol{e}] := [\boldsymbol{x}] + [\boldsymbol{r}]$ and send their shares to the first party P_1.

3. P_1 reconstructs the secrets $\boldsymbol{e}$ and computes $\tilde{\boldsymbol{e}} = F_p(\boldsymbol{e})$. P_1 generates a random degree-t packed Shamir sharing $[\tilde{\boldsymbol{e}}]$ and distributes the shares to other parties.

4. All parties locally compute $[\tilde{\boldsymbol{x}}] := [\tilde{\boldsymbol{e}}] - [\tilde{\boldsymbol{r}}]$.

To see the correctness, note that in the second step we have $\boldsymbol{e} = \boldsymbol{x} + \boldsymbol{r}$. Therefore,

$$\tilde{\boldsymbol{x}} = F_p(\boldsymbol{x}) = F_p(\boldsymbol{e} - \boldsymbol{r}) = F_p(\boldsymbol{e}) - F_p(\boldsymbol{r}) = \tilde{\boldsymbol{e}} - \tilde{\boldsymbol{r}}.$$

The communication complexity of this protocol is $O(n/k)$ elements per secret (excluding the cost for the preparation of $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$).

As noted in [DIK10], the main issue of this approach is how to *efficiently* prepare a pair of random sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$. Although there are known techniques to prepare random sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ for *a fixed* permutation p such that the amortized communication complexity per pair is $O(n)$ elements where in turn the amortized cost per secret is $O(n/k)$ elements, these techniques suffer a large overhead (at least $O(n^2)$ elements) that is independent of the number of sharings we want to prepare. It means that the overhead of preparing random sharings depends on the number of different permutations we want to perform. In the worst case where each time we need to perform a different permutation, the overhead of each pair of random sharings is as large as $O(n^2)$ elements, which eliminates the benefit of using the packed Shamir sharing scheme. In [DIK10], this issue is solved by compiling the circuit such that only $O(\log n)$ different permutations are needed in the computation with the cost of blowing up the circuit size by a factor of $O(\log |C|)$, where $|C|$ is the circuit size. This approach does not achieve our goal since the amortized communication complexity per gate becomes $O(\log |C| \cdot n/k)$ elements. To generate random sharings for m permutations, our idea is to first generate random sharings for a limited number ($O(n^2)$) of different permutations which are related to the input permutations, and then transform them to the random sharings for the desired permutations (the input permutations). In this way, since we only need to prepare random sharings for $O(n^2)$ different permutations, we do not suffer the quadratic overhead in the communication complexity even if all the input permutations are different. Moreover, we do not need to compile the circuit and therefore do not suffer the $O(\log |C|)$ factor in the communication complexity as that in [DIK10]. As a result, the amortized communication complexity of our permutation protocol is $O(n/k)$ elements per secret.

Before introducing our idea, we first introduce a useful functionality $\mathcal{F}_{\text{select}}$, which selects secrets from one or more packed Shamir sharings and outputs a single sharing which contains the chosen secrets. Later on, we will use $\mathcal{F}_{\text{select}}$ to solve the above issue of preparing random sharings for permutations. Concretely, $\mathcal{F}_{\text{select}}$ takes as input k degree-t packed Shamir sharings $\{[\boldsymbol{x}^{(i)}]\}_{i=1}^{k}$ (which do not need to be distinct) and outputs a degree-t packed Shamir sharing of $\boldsymbol{y}$ such that $y_i = x_i^{(i)}$. Effectively, $\mathcal{F}_{\text{select}}$ chooses the i-th secret of $[\boldsymbol{x}^{(i)}]$ and generates a new degree-t packed Shamir sharing $[\boldsymbol{y}]$ that contains the chosen secrets. Note that *the secrets we choose are from different positions and the positions of these secrets remain unchanged in the output sharing.* To realize $\mathcal{F}_{\text{select}}$, we observe that $\boldsymbol{y}$ can be computed by $\sum_{i=1}^{k} \boldsymbol{e}^{(i)} * \boldsymbol{x}^{(i)}$, where $\boldsymbol{e}^{(i)}$ is a constant vector where the i-th entry is 1 and all other entries are 0, and $*$ denotes the coordinate-wise multiplication operation. We realize $\mathcal{F}_{\text{select}}$ by extending the basic protocol for multiplication gates as described in Section 4.2. The amortized communication complexity of $\mathcal{F}_{\text{select}}$ is $O(n/k)$ elements per secret.

Using $\mathcal{F}_{\text{select}}$ to Generate Random Sharings for Permuting Secrets. For all $i, j \in \{1, 2, \ldots, k\}$, we say a pair of degree-t packed Shamir sharings $([\boldsymbol{x}], [\boldsymbol{y}])$ contains an (i, j)-component if $x_i = y_j$. To perform a permutation $p(\cdot)$, we need to prepare two random degree-t packed Shamir sharings $([\boldsymbol{r}], [F_p(\boldsymbol{r})])$. We can view $([\boldsymbol{r}], [F_p(\boldsymbol{r})])$ as a composition of an $(i, p(i))$-component for all $i \in [k]$.

Now we introduce a new approach for preparing random sharings $([\boldsymbol{r}], [F_p(\boldsymbol{r})])$:

1. Let $q_1, q_2, \ldots, q_k$ be k different permutations over $\{1, 2, \ldots, k\}$ such that for all $i \in [k]$, $q_i(i) = p(i)$.

2. All parties prepare a pair of random sharings for each permutation q_i, denoted by $([\boldsymbol{r}^{(i)}], [F_{q_i}(\boldsymbol{r}^{(i)})])$. Since $q_i(i) = p_i$, $([\boldsymbol{r}^{(i)}], [F_{q_i}(\boldsymbol{r}^{(i)})])$ contains an $(i, p(i))$-component.

3. To prepare $([\boldsymbol{r}], [F_p(\boldsymbol{r})])$, we can use $\mathcal{F}_{\text{select}}$ to select the $(i, p(i))$-component from $([\boldsymbol{r}^{(i)}], [F_{q_i}(\boldsymbol{r}^{(i)})])$ for all $i \in [k]$. More concretely, for $[\boldsymbol{r}]$, we use $\mathcal{F}_{\text{select}}$ to select the i-th secret of $[\boldsymbol{r}^{(i)}]$ for all $i \in [k]$. For $[F_p(\boldsymbol{r})]$, we use $\mathcal{F}_{\text{select}}$ to select the $p(i)$-th secret of $[F_{q_i}(\boldsymbol{r}^{(i)})]$ for all $i \in [k]$.

While this way of preparing a single pair of random sharings for the permutation p requires k pairs of random sharings for k permutations $q_1, \ldots, q_k$, we note that *the unused components of* $([\boldsymbol{r}^{(i)}], [F_{q_i}(\boldsymbol{r}^{(i)})])$ can potentially be used to prepare random sharings for other permutations.

In general, when we want to prepare random sharings for m permutations $p_1(\cdot), p_2(\cdot), \ldots, p_m(\cdot)$, relying on $\mathcal{F}_{\text{select}}$, it is sufficient to alternatively prepare random sharings for m permutations $q_1(\cdot), q_2(\cdot), \ldots, q_m(\cdot)$ such that:

- For all $i, j \in \{1, 2, \ldots, k\}$, the number of permutations $p \in \{p_1, p_2, \ldots, p_m\}$ which satisfies that $p(i) = j$ is equal to the number of permutations $q \in \{q_1, q_2, \ldots, q_m\}$ which satisfies that $q(i) = j$.

Then, from $i = 1$ to m, a pair of random sharings for the permutation p_i can be prepared by using $\mathcal{F}_{\text{select}}$ to choose the first unused $(j, p_i(j))$-component for all $j \in [k]$.

The major benefit of this approach is that *we can limit the number of different permutations in* $\{q_1, q_2, \ldots, q_m\}$ as we show in Theorem 2.

Theorem 2. *Let $m, k \geq 1$ be integers. For all m permutations $p_1, p_2, \ldots, p_m$ over $\{1, 2, \ldots, k\}$, there exists m permutations $q_1, q_2, \ldots, q_m$ over $\{1, 2, \ldots, k\}$ such that:*

- *For all $i, j \in \{1, 2, \dots, k\}$, the number of permutations $p \in \{p_1, p_2, \dots, p_m\}$ such that $p(i) = j$ is the same as the number of permutations $q \in \{q_1, q_2, \dots, q_m\}$ such that $q(i) = j$.*
- *$q_1, q_2, \dots, q_m$ contain at most k^2 different permutations.*

Moreover, $q_1, q_2, \dots, q_m$ can be found within polynomial time given $p_1, p_2, \dots, p_m$.

Recall that the issue of using known techniques to prepare random sharings for $p_1, p_2, \dots, p_m$ is that there will be an overhead of $O(n^2)$ elements per different permutation in $p_1, p_2, \dots, p_m$. Relying on $\mathcal{F}_{\text{select}}$, we only need to prepare random sharings for permutations $q_1, \dots, q_m$, which contain at most $k^2 \leq n^2$ different permutations. In this way, the overhead is independent of the number of permutations and the circuit size. Recall that the amortized communication complexity for each pair of random sharings is $O(n/k)$ elements per secret, and our protocol for $\mathcal{F}_{\text{select}}$ and the permutation protocol from [DIK10] also have the same amortized communication complexity, i.e., $O(n/k)$ elements per secret. Therefore, the overall communication complexity to perform an arbitrary permutation on the secrets of a single secret sharing is $O(n/k)$ elements per secret.

Using Hall's Marriage Theorem to Prove Theorem 2. We note that Theorem 2 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 $\{1, 2, \dots, k\}$ corresponds to a perfect matching in a bipartite graph: the set of vertices are $V_1 = V_2 = \{1, 2, \dots, k\}$, and the set of edges are $\mathcal{E} = \{(i, p(i))\}_{i=1}^{k}$.

We first construct a bipartite graph $G = (V_1, V_2, E)$ where $V_1 = V_2 = \{1, 2, \dots, k\}$ and E contains all edges in the perfect matching that $p_1, p_2, \dots, p_m$ correspond to. Strictly speaking, G is a multi-graph since a pair of vertices may have multiple edges. Note that Theorem 2 is equivalent to decomposing G into m perfect matching such that the number of different perfect matching is bounded by k^2. Our idea of finding these m perfect matching is to repeat the following steps until E becomes empty:

1. We first find a perfect matching $\mathcal{E} \subset E$ in G.
2. We repeatedly remove $\mathcal{E}$ from E until $\mathcal{E}$ is no longer a subset of E. The number of times that $\mathcal{E}$ is removed from E is the number of times that $\mathcal{E}$ appears in the output perfect matching.

Note that the number of different perfect matches is the same as the number of iterations of the above two steps. Suppose the first step always succeeds. The second step guarantees that in each iteration, we will completely use up the edges between one pair of vertices in E. Since there are at most k^2 different pairs of vertices, the above process will terminate within k^2 iterations.

For the first step, we use Hall's Marriage Theorem to prove the existence of a perfect matching.

Theorem 3 (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 at the beginning of each iteration 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 1. *There exists a perfect matching in a d-regular bipartite graph.*

Therefore, it is sufficient to show that the graph G at the beginning of each iteration is a d-regular bipartite graph. Recall that in the beginning, the set of edges E contains all edges in the perfect matching that $p_1, p_2, \ldots, p_m$ correspond to. Since by definition, the degree of each vertex in a perfect matching is exactly 1, the degree of each vertex in G is m, which means that G is a m-regular bipartite graph. In each iteration, we first find a perfect matching in Step 1 and then repeatedly remove this perfect matching from E in Step 2. Each time of removing a perfect matching reduces the degree of each vertex in G by 1. Thus, G is still a d-regular bipartite graph after each remove of a perfect matching. Therefore, the graph G at the beginning of each iteration is a d-regular bipartite graph.

2.3 Obtaining Input Sharings for Multiplication Gates and Addition Gates

So far, we have introduced how to perform a permutation to the secrets of a single sharing to obtain the correct order. However, this only solves the problem when we have all the values we want in a single sharing. During the computation, such a sharing does not come for free since the values we want may be scattered in one or more output sharings of previous layers. This requires us to collect the secrets from those sharings and generate a single sharing for these secrets efficiently.

Our starting point is the functionality $\mathcal{F}_{\text{select}}$. Recall that $\mathcal{F}_{\text{select}}$ allows us to select secrets from one or more sharings and generate a new sharing for the chosen secrets if the secrets we select are *in different positions*. To use $\mathcal{F}_{\text{select}}$, we consider what we call the *non-collision* property stated in Property 1.

Property 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.*

Note that if we can guarantee the non-collision property, then we can use $\mathcal{F}_{\text{select}}$ to generate the input sharing we want. 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).

Note that this requirement can be met without loss of generality 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. In the next subsection, we will discuss how to evaluate fan-out gates efficiently. 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 4. *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 4 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. Therefore, we can use $\mathcal{F}_{\text{select}}$ to choose these secrets and obtain the desired input sharing.

Using Hall's Marriage Theorem to Prove Theorem 4. Let $\boldsymbol{N}$ be the matrix in Theorem 4. 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 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. To prove the existence of a perfect matching, we show that the graph G is a k-regular bipartite graph and rely on the corollary (Corollary 1) of Hall's Marriage Theorem. 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 1, there exists a perfect matching in G. The same arguments work for other iterations. We refer the readers to Section 4.3 for more details.

It is worth noting that we use Hall's Marriage Theorem to solve two different problems:

- In Theorem 2, we use Hall's Marriage Theorem to find a different set of permutations $q_1, q_2, \ldots, q_m$ given the permutations $p_1, p_2, \ldots, p_m$ and limit the number of different permutations in $q_1, q_2, \ldots, q_m$.

- In Theorem 4, we use Hall's Marriage Theorem to find a permutation for each output sharing to achieve the non-collision property (Property 1).

2.4 Handling Fan-out Gates

We briefly discuss how to evaluate fan-out gates efficiently. We first model the problem as follows: given a degree-t packed Shamir sharing $[\boldsymbol{x}]$ 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-t 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 Section 5.1 for how we handle the edge case.)

Our idea is to compute the output sharings one by one. For each output sharing $[\boldsymbol{y}]$, all values of $\boldsymbol{y}$ come from $\boldsymbol{x}$, which means that we may write $\boldsymbol{y}$ as a linear function of $\boldsymbol{x}$. Let F be a linear map such that $\boldsymbol{y} = F(\boldsymbol{x})$. To compute $[\boldsymbol{y}]$, we can prepare a pair of random sharings $([\boldsymbol{r}], [F(\boldsymbol{r})])$ and use the same method to compute $[\boldsymbol{y}]$ as that for permutations. Then we face the same problem that naively preparing the random sharings $([\boldsymbol{r}], [F(\boldsymbol{r})])$ suffer an overhead which depends on the number of different linear maps F. In the worst case where we need a different linear map for different output packed Shamir sharing, the overhead of preparing each pair of random sharings is as large as $O(n^2)$ elements, which eliminate the benefit of using the packed Shamir sharing scheme.

We follow the same idea as that for permutation to prepare the random sharings $([\boldsymbol{r}], [F(\boldsymbol{r})])$: Given m different linear maps $F_1, F_2, \dots, F_m$, we will prepare random sharings for m other linear maps $G_1, G_2, \dots, G_m$ and then recompose the components in the random sharings for $G_1, G_2, \dots, G_m$ to obtain random sharings for $F_1, F_2, \dots, F_m$. The main difficulty is that it is unclear how to define a component. Our solution includes the following additional steps:

- We require the secrets of the output packed Shamir sharings to be in a specific order.
- To compute each output packed Shamir sharing $[\boldsymbol{y}]$, we first permute the secrets of $[\boldsymbol{x}]$ based on $\boldsymbol{y}$.

These two steps allow us to properly define a component in a way that we can efficiently find $G_1, G_2, \dots, G_m$ such that the above idea works. We refer the readers to Section 4.4 for more details.

2.5 Overview of Our Semi-honest Protocol

So far, we have introduced all the building blocks we need in our semi-honest protocol. To evaluate a single circuit:

1. All parties first transform the circuit to a good form in the sense that the number of gates of each type in each layer is a multiple of k. The transformation is done locally by running a deterministic algorithm. Unlike [DIK10], our transformation only increases the circuit size by a constant factor and an additive term $O(k \cdot \mathsf{Depth})$, where the latter term comes from the fact that the number of gates in each layer is a multiple of k after the transformation. The same term (or a larger term) also exists in [DIK10, GIP15]. We refer the readers to Section 5.1 for more details.
2. All parties preprocess the circuit to determine how the wire values should be packed. Also, all parties compute a permutation for each output sharing for the non-collision property (see Property 1 in Section 2.3). This step is also done locally. We refer the readers to Section 5.2 for more details.
3. Finally, all parties evaluate the circuits using the protocols we described above. We refer the readers to Section 5.3 for more details.

Note that only the third step requires communication. We briefly analyze the communication complexity. For each group of k gates, all parties use the basic protocol to evaluate these gates. The communication complexity of the basic protocol is $O(n)$ elements. To prepare the input sharings for this group of k (addition, multiplication, or output) gates, we need to evaluate fan-out gates, perform permutations to achieve the non-collision property, use $\mathcal{F}_{\mathrm{select}}$ to collect the secrets of the input sharings, and perform permutations again to obtain the correct orders. Since each operation requires $O(n)$ elements, the amortized communication complexity per gate is $O(n/k)$ elements.

2.6 Achieving Malicious Security

We briefly discuss how to compile our semi-honest protocol to a fully malicious one. Our main observation is that most of our semi-honest protocols have already achieved perfect privacy *against a fully malicious adversary*, namely the executions of these protocols do not leak any information to the adversary. Also, the deviation of a fully malicious adversary can be reduced to the following two kinds of attacks:

- An adversary can distribute an inconsistent degree-t packed Shamir sharing.
- An adversary can add additive errors to the secrets of the output sharing.

To achieve malicious security, our idea is to first run our semi-honest protocol before the output phase, check whether the above two kinds of attacks are launched by the adversary, and finally reconstruct the output.

To this end, for each semi-honest protocol, we first construct a functionality which allows the adversary to launch the above two kinds of attacks, and prove that our semi-honest protocol securely (with abort) computes the new functionality against a fully malicious adversary (see Section 6 for more details). Then we construct protocols to check whether the above two kinds of attacks are launched by the adversary. We view the computation as a composition of two parts: (1) evaluation of the basic gates, i.e., addition gates and multiplication gates, and (2) network routing, i.e., computing input sharings of each layer using the output sharings from previous layers.

- For the first part, since addition gates are computed without interaction, it is sufficient to only check the correctness of multiplications. We extend the recent sub-linear verification techniques [BBCG$^+$19, GSZ20] which are used in the honest majority setting (i.e., the corruption threshold $t' = t$) to our setting (i.e., the corruption threshold $t' = t - k + 1$). We refer the readers to Section 7.1 for more details.

- For the second part, it includes evaluating fan-out gates, performing permutations to achieve the non-collision property, using $\mathcal{F}_{\text{select}}$ to collect the secrets of the input sharings, performing permutations again to obtain the correct orders. We note that the network routing does not change the secret values. Instead, its goal is to create new sharings which contain the secret values we want in the correct positions. Thus, it is sufficient to only focus on the front sharing before the network routing and the end sharing after the network routing, and check whether they have the same values. We refer the readers to Section 7.2 for more details.

Finally, when both checks pass, all parties reconstruct the output as the semi-honest protocol.

Remark 1. *We note that the multiplication protocol is an exception in the sense that it cannot be reduced directly to the additive attacks we mention above. In fact, the work [GIP15] showed that a malicious attack can only be reduced to a linear attack, where the error in the output secret can depend on the input secrets. Our observation is that the linear attack is due to the inconsistency of the input sharings. If the input sharings are consistent, then the linear attack in [GIP15] degenerates to an additive attack to the final result. To model such a security property, we use a weaker functionality for the multiplication protocol, which does not guarantee the correctness of the multiplication result when the input sharings are inconsistent. The verification is done by first checking the consistency of all sharings. If the verification passes, then the attack of an adversary degenerates to additive attacks, which allows us to use the efficient verification protocol for multiplication gates in previous works. More discussion can be found in Section 6.1 for the multiplication protocol, and Section 7.1 for the verification protocol.*

3 Preliminaries

3.1 The Model

In this work, we use the *client-server* model for the secure multi-party computation. 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 $n = 2t + 1$ 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 by the number of bits via private channels.

We focus on functions that can be represented as arithmetic circuits over a finite field $\mathbb{F}$ with input, addition, multiplication, and output gates. We use κ to denote the security parameter, C to denote the circuit, and $|C|$ for the size of the circuit. We assume that the field size is $|\mathbb{F}| \geq 2n$.

3.1.1 Security Definition.

Let $1 \leq k \leq t$ be an integer. Let $\mathcal{F}$ be a secure function evaluation functionality. An adversary $\mathcal{A}$ can corrupt at most c clients and $t' = t - k + 1$ servers, provide inputs to corrupted clients, and receive all messages sent to corrupted clients and servers. In this work, we consider both semi-honest adversaries and fully malicious adversaries.

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

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

Ideal-World Execution. In the ideal world, a simulator `Sim` simulates honest clients and servers and interacts with the adversary $\mathcal{A}$. Furthermore, `Sim` has one-time access to $\mathcal{F}$, which includes providing inputs of corrupted clients and servers to $\mathcal{F}$, receiving the outputs of corrupted clients and servers, 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 clients and servers and the view of the adversary.

We say that a protocol π securely computes $\mathcal{F}$ if there exists a simulator `Sim`, such that for all adversary $\mathcal{A}$, the distribution of the output of the real-world execution is *statistically indistinguishable* from the distribution of the output of the ideal-world execution. We refer the readers to [GIP$^+$14] for a formal definition.

3.1.2 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 $\mathcal{C}$ 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 $t' = t - k + 1$ parties. At a high-level, for an adversary $\mathcal{A}$ which controls $t' < t - k + 1$ parties, we may construct another adversary $\mathcal{A}'$ which controls additional $(t - k + 1) - t'$ parties and behaves as follows:

- For a party corrupted by $\mathcal{A}$, $\mathcal{A}'$ follows the structure 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 $(t-k+1) - t'$ 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. In the following, we assume that there are exactly $t' = t - k + 1$ corrupted parties.

3.2 Secret Sharing Scheme

Packed Shamir Secret Sharing Scheme. In this work, we will use the packed secret-sharing technique introduced by Franklin and Yung [FY92]. This is a generalization of the standard Shamir secret sharing scheme [Sha79]. Let n be the number of parties and k be the number of secrets that are packed in one sharing. Let $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_k$ be $n + k$ distinct non-zero 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 $(w_1, \ldots, w_n)$ which satisfies that, there exists a polynomial $f(\cdot) \in \mathbb{F}[X]$ of degree at most d such that $\forall i \in [k], f(\beta_i) = x_i$ and $\forall i \in [n], f(\alpha_i) = w_i$. 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 $\boldsymbol{x}$.

We will use $[\boldsymbol{x}]$ to denote a degree-t packed Shamir sharing of $\boldsymbol{x} \in \mathbb{F}^k$, and $\langle \boldsymbol{x} \rangle$ to denote a degree-$2t$ packed Shamir sharing. Recall that the number of corrupted parties is at most $t-k+1$. Therefore, using degree-t packed Shamir sharings is sufficient to protect the privacy of the secrets. 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 $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\boldsymbol{x}+\boldsymbol{y}] = [\boldsymbol{x}] + [\boldsymbol{y}]$.
- Multiplication: Let $*$ denote coordinate-wise multiplication. For all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $\langle \boldsymbol{x} * \boldsymbol{y} \rangle = [\boldsymbol{x}] \cdot [\boldsymbol{y}]$.

These two properties directly follow from the computation of the polynomials.

For a constant vector $\boldsymbol{v} \in \mathbb{F}^k$ which is known by all parties, sometimes it is convenient to transform it to a degree-t packed Shamir sharing. This can be done by constructing a polynomial $f(\cdot) \in F[X]$ of degree $k-1$ such that for all $i \in [k]$, $f(\beta_i) = v_i$. The i-th share of $[\boldsymbol{v}]$ is defined to be $f(\alpha_i)$ as usual.

Abstract General Linear Secret Sharing Schemes. We adopt the notion of an abstract definition of a general linear secret sharing scheme (GLSSS) in [CCXY18]. The following notations are borrowed from [CCXY18].

For non-empty sets U and $\mathcal{I}$, $U^{\mathcal{I}}$ denotes the indexed Cartesian product $\prod_{i \in \mathcal{I}} U := \{f : \mathcal{I} \to U\}$. When $\mathcal{I}$ is a finite set, one may think that each element in $\prod_{i \in \mathcal{I}} U$ is a vector of dimension $|\mathcal{I}|$ where each entry is an element in U associated with a unique index in $\mathcal{I}$. 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 $(u_i)_{i \in A} \in U^A$. Let K be a field.

Definition 1 (Abstract K-GLSSS [CCXY18])**.** *A general K-linear secret sharing scheme Σ consists of the following data:*

- *A set of parties $\mathcal{I} = \{1, \ldots, n\}$*
- *A finite-dimensional K-vector space Z, the secret space.*
- *A finite-dimensional K-vector space U, the share space.*
- *A K-linear subspace $C \subset U^{\mathcal{I}}$, where the latter is considered a K-vector space in the usual way (i.e., direct sum).*
- *A surjective K-linear map $\Phi : C \to Z$, its defining map.*

Definition 2 ([CCXY18])**.** *Suppose $A \subset \mathcal{I}$ is nonempty. Then A is a privacy set if the K-linear map*

$$(\Phi, \pi_A) : C \longrightarrow Z \times \pi_A(C), \quad x \mapsto (\Phi(x), \pi_A(x))$$

is surjective. Finally, A is a reconstruction set if, for all $x \in C$, it holds that

$$\pi_A(x) = 0 \Rightarrow \Phi(x) = 0.$$

We give a high-level explanation of why Definition 2 accords with the intuitive definitions of privacy sets and reconstruction sets. We say A is a privacy set if the shares of parties in A are independent of the secret. In Definition 2, the K-linear map (Φ, π_A) maps each sharing in C to its secret and the shares of parties in A. When this map is surjective, the secret can take all possible values in the secret space no matter what the shares of parties in A are, which means that the secret is independent of the shares of parties in A.

We say A is a reconstruction set if the shares of parties in A fully determine the secret. In other words, if two different sharings have the same shares for parties in A, they should have the same secret. For two different sharings $y, y' \in C$, we have

$$\pi_A(y) = \pi_A(y') \Rightarrow \pi_A(y - y') = 0 \Rightarrow \Phi(y - y') = 0 \Rightarrow \Phi(y) = \Phi(y'),$$

where the second step follows from Definition 2 and the last equation shows that the secrets of y and y' are the same.

3.3 Useful Building Blocks

In this part, we will introduce two functionalities that will be used in our main construction.

- The first functionality $\mathcal{F}_{\text{coin}}$ allows all parties to generate a random element.
- The second functionality $\mathcal{F}_{\text{rand}}$ allows all parties to prepare a random sharing for a given $\mathbb{F}$-linear secret sharing scheme.

Generating Random Coins The first functionality $\mathcal{F}_{\text{coin}}(\mathbb{F})$ allows all parties to generate a random field element in $\mathbb{F}$. The description of $\mathcal{F}_{\text{coin}}(\mathbb{F})$ appears in Functionality 1.

Functionality 1: $\mathcal{F}_{\text{coin}}(\mathbb{F})$

1. $\mathcal{F}_{\text{coin}}$ samples a random field element r in $\mathbb{F}$.
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.

An instantiation of this functionality can be found in [GSZ20] (Protocol 6 in Section 3.5 of [GS20]), which has communication complexity of $O(n^2)$ elements in $\mathbb{F}$. The original protocol is secure when up to t parties are corrupted. Therefore, it is also secure in our case where the number of corrupted parties is bounded by $t' = t - k + 1$.

Preparing Random Sharings for $\mathbb{F}$-GLSSS In this part, we introduce a functionality $\mathcal{F}_{\text{rand}}$, which comes from [PS21]. It allows all parties to prepare a random sharing for a given $\mathbb{F}$-linear secret sharing scheme Σ. Let $[\![x]\!]$ denote a sharing in Σ of secret x. For a set $A \subset \mathcal{I}$, recall that $\pi_A([\![x]\!])$ refers to the shares of $[\![x]\!]$ held by parties in A. We assume that Σ satisfies the following property:

- Given a set $A \subset \mathcal{I}$ and a set of shares $\{a_i\}_{i \in A}$ for parties in A, let

$$\Sigma(A, (a_i)_{i \in A}) := \{[\![x]\!] \mid [\![x]\!] \in \Sigma \text{ and } \pi_A([\![x]\!]) = (a_i)_{i \in A}\}.$$

  Then, there is an efficient algorithm which outputs that either $\Sigma(A, (a_i)_{i \in A}) = \emptyset$, or a random sharing $[\![x]\!]$ in $\Sigma(A, (a_i)_{i \in A})$.

The description of the functionality $\mathcal{F}_{\text{rand}}$ appears in Functionality 2. In short, $\mathcal{F}_{\text{rand}}$ allows the adversary to specify the shares held by corrupted parties. Based on these shares, $\mathcal{F}_{\text{rand}}$ generates a random sharing in Σ and distributes the shares to honest parties. Note that, when the set of corrupted parties is a privacy set, the secret is independent of the shares chosen by the adversary.

In [PS21], Polychroniadou and Song proposed an instantiation of $\mathcal{F}_{\text{rand}}$ which is secure against an adversary that corrupts t parties. We note that we can easily modify their protocol to achieve security against an adversary that corrupts $t - k + 1$ parties. For completeness, we provide the details of the protocol and the proof in Appendix A. Suppose the share size of a sharing in σ is sh field elements in $\mathbb{F}$. The communication complexity of generating N random sharings in Σ is $O(N \cdot n \cdot \mathsf{sh} + n^3 \cdot \kappa)$ elements in $\mathbb{F}$.

3.4 Permutation Matrix, Bipartite Graph and Hall's Marriage Theorem

Definition 3 (Permutation Matrix)**.** *Let $k \geq 1$ be an integer. A matrix $\boldsymbol{M} \in \{0,1\}^{k \times k}$ is a permutation matrix if for each row and each column, there is exactly one entry which is* 1.

Functionality 2: $\mathcal{F}_{\text{rand}}$

1. $\mathcal{F}_{\text{rand}}$ receives from the adversary the set of corrupted parties, denoted by $\mathcal{C}$, and a set of shares $(s_i)_{i\in\mathcal{C}}$ such that $\Sigma(\mathcal{C},(s_i)_{i\in\mathcal{C}}) \neq \emptyset$. Then $\mathcal{F}_{\text{rand}}$ randomly samples $[\![r]\!] \in \Sigma(\mathcal{C},(s_i)_{i\in\mathcal{C}})$.
2. $\mathcal{F}_{\text{rand}}$ asks the adversary whether it should continue or not.
    - If the adversary replies `abort`, $\mathcal{F}_{\text{rand}}$ sends `abort` to honest parties.
    - If the adversary replies `continue`, for each honest party P_i, $\mathcal{F}_{\text{rand}}$ sends the i-th share of $[\![r]\!]$ to P_i.

For a permutation $p(\cdot)$ over $\{1,2,\ldots,k\}$, let $\boldsymbol{M}_p$ be a permutation matrix such that for all $i,j \in \{1,\ldots,k\}$, $(\boldsymbol{M}_p)_{i,j} = 1$ iff $p(i) = j$. Note that for each permutation matrix $\boldsymbol{M}'$, there exists a permutation $p(\cdot)$ such that $\boldsymbol{M}' = \boldsymbol{M}_p$.

Definition 4 (Balanced Matrix)**.** *Let $k \geq 1$ be an integer. A matrix $\boldsymbol{M} \in \mathbb{N}^{k\times k}$ is a balanced matrix if for each row and each column, the summation of all the entries is the same.*

Note that for all permutations $p(\cdot)$ over $\{1,2,\ldots,k\}$, the permutation matrix $\boldsymbol{M}_p$ is a balanced matrix since the summation of the entries in each row and each column is 1.

Definition 5 (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 6 (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 3 (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 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 $|N(V_1')| \geq |V_1'|$. □

Corollary 2. *Let $k \geq 1$ be an integer. For all non-zero balanced matrix $\boldsymbol{N} \in \mathbb{N}^{k\times k}$, there exists a permutation matrix $\boldsymbol{M}$ such that for all $i,j \in \{1,2,\ldots,k\}$, $\boldsymbol{N}_{i,j} \geq \boldsymbol{M}_{i,j}$.*

Proof. For a matrix $\boldsymbol{N} \in \mathbb{N}^{k\times k}$, we can map it to a bipartite graph $G = (V_1,V_2,E)$, where $V_1 = V_2 = \{1,2,\ldots,k\}$ and the number of edges between $(i,j) \in V_1 \times V_2$ is $\boldsymbol{N}_{i,j}$.

Therefore, a non-zero balanced matrix $\boldsymbol{N} \in \mathbb{N}^{k\times k}$ maps to a regular bipartite graph G, and a permutation matrix $\boldsymbol{M}$ maps to a perfect matching in G. According to Corollary 1, there exists a perfect matching in G, which can map back to a permutation matrix $\boldsymbol{M}$. Note that for all $i,j \in \{1,2,\ldots,k\}$, $\boldsymbol{M}_{i,j}$ is 1 iff (i,j) is in the perfect matching, which is an edge in G, implying that $\boldsymbol{N}_{i,j} \geq 1$. Therefore, for all $i,j \in \{1,2,\ldots,k\}$, $\boldsymbol{N}_{i,j} \geq \boldsymbol{M}_{i,j}$. □

4 Circuit Evaluation - Against a Semi-honest Adversary

In this section, we discuss how to evaluate a general circuit by using the packed Shamir sharing scheme. For simplicity, we assume the adversary is semi-honest. Here is a high-level structure of this section.

1. We start with the basic protocols to evaluate input gates, addition gates, multiplication gates, and output gates using the packed Shamir sharing scheme. These protocols are simple variants of the protocols in [DN07], which focuses on the adversary that can corrupt t parties.

2. To use these protocols to evaluate addition gates and multiplication gates, we need the secrets in the input packed Shamir sharings to have the correct order. Assuming each input sharing contains all the secret values we want, we discuss how to permute the secrets in each input sharing to the correct order.

3. Next, we show how to collect the secrets of an input packed Shamir sharing from the output sharings of previous layers. Our solution requires that each output wire from each layer is only used once in the computation, as an input wire to a single layer. This requirement can be met by further requiring that there is a fan-out gate right after each gate that copies the output wire the number of times it is used in later layers.

4. Finally, we discuss how to evaluate fan-out gates efficiently.

Recall that we are in the client-server model where there are c clients and $n = 2t + 1$ parties (servers). Recall that $1 \leq k \leq t$ is an integer. An adversary is allowed to corrupt $t' = t - k + 1$ parties. We will use the degree-t packed Shamir sharing scheme, which can store k secrets within one sharing. Recall that $\mathcal{C}$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

4.1 Basic Protocols for Input Gates, Addition Gates, Multiplication Gates, and Output Gates

We distinguish input gates and output gates belonging to different clients. For each client, we assume the number of input gates belonging to this client and the number of output gates belonging to this client are multiples of k. For each layer, we assume that the number of addition gates and the number of multiplication gates are multiples of k. In Section 5, we will show how to compile a general circuit to meet this requirement.

Evaluating Input Gates and Output Gates. The functionalities $\mathcal{F}_{\text{input-semi}}$ and $\mathcal{F}_{\text{output-semi}}$ are described in Functionality 3 and Functionality 4 respectively.

Functionality 3: $\mathcal{F}_{\text{input-semi}}$

1. Suppose $\boldsymbol{x} \in \mathbb{F}^k$ is the input associated with the input gate which belongs to the Client. $\mathcal{F}_{\text{input-semi}}$ receives the input $\boldsymbol{x}$ from the Client.

2. $\mathcal{F}_{\text{input-semi}}$ receives from the adversary a set of shares $\{s_i\}_{i \in \mathcal{C}}$. $\mathcal{F}_{\text{input-semi}}$ samples a random degree-t packed Shamir sharing $[\boldsymbol{x}]$ such that for all $P_i \in \mathcal{C}$, the i-th share of $[\boldsymbol{x}]$ is s_i.

3. $\mathcal{F}_{\text{input-semi}}$ distributes the shares of $[\boldsymbol{x}]$ to honest parties.

For input gates, the client who holds the input $\boldsymbol{x} \in \mathbb{F}^k$ generates a random degree-t packed Shamir sharing and distributes the shares to all parties. For output gates, all parties send their shares to the client who should receive the results to allow the client to reconstruct the outputs. The protocols Input and Output appear in Protocol 5 and Protocol 6 respectively. The communication complexity of each protocol is $O(n)$ field elements.

Functionality 4: $\mathcal{F}_{\text{output-semi}}$

1. Suppose $[\boldsymbol{x}]$ is the sharing associated with the output gate which belongs to the Client. $\mathcal{F}_{\text{output-semi}}$ receives the shares of $[\boldsymbol{x}]$ from honest parties.

2. $\mathcal{F}_{\text{output-semi}}$ recovers the whole sharing $[\boldsymbol{x}]$, and sends the shares of corrupted parties to the adversary.

3. $\mathcal{F}_{\text{output-semi}}$ reconstructs $\boldsymbol{x}$ and sends it to the Client.

Protocol 5: INPUT

1. Suppose $\boldsymbol{x} \in \mathbb{F}^k$ is the input associated with the input gate which belongs to the Client. The Client generates a random degree-t packed Shamir sharing $[\boldsymbol{x}]$.

2. The Client distributes the shares of $[\boldsymbol{x}]$ to all parties.

Protocol 6: OUTPUT

1. Suppose $[\boldsymbol{x}]$ is the sharing associated with the output gate which belongs to the Client. All parties send their shares of $[\boldsymbol{x}]$ to the Client.

2. The Client reconstructs the result $\boldsymbol{x}$ from the shares of $[\boldsymbol{x}]$.

Lemma 1. *Protocol* INPUT *securely computes $\mathcal{F}_{\textit{input-semi}}$ against a semi-honest adversary who controls $t' = t - k + 1$ parties.*

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

If the Client is corrupted, $\mathcal{S}$ receives the shares of honest parties from the Client. Note that there are $n - t' = t + k \geq t + 1$ honest parties. $\mathcal{S}$ uses the shares of honest parties to recover the whole sharing $[\boldsymbol{x}]$. $\mathcal{S}$ sends the secrets $\boldsymbol{x}$ to $\mathcal{F}_{\text{input-semi}}$ on behalf of the Client. Then $\mathcal{S}$ sends the shares of $[\boldsymbol{x}]$ held by corrupted parties to $\mathcal{F}_{\text{input-semi}}$. Note that a degree-$t$ packed Shamir sharing is determined by the shares of corrupted parties and the secrets, which are $t' + k = t + 1$ shares and secrets. Therefore, the sharing generated by $\mathcal{F}_{\text{input-semi}}$ is identical to $[\boldsymbol{x}]$.

If the Client is honest, $\mathcal{S}$ generates t' random elements in $\mathbb{F}$ as the shares of $[\boldsymbol{x}]$ held by corrupted parties and distributes them to corrupted parties on behalf of the Client. Note that these shares have the same distribution as the shares generated by the Client in the real world. $\mathcal{S}$ sends the shares of $[\boldsymbol{x}]$ of corrupted parties to $\mathcal{F}_{\text{input-semi}}$. Since the whole sharing $[\boldsymbol{x}]$ is determined by the shares of corrupted parties and the secrets, and the shares of corrupted parties in both worlds have the same distribution, the distributions of $[\boldsymbol{x}]$ in both worlds are identical. □

Lemma 2. *Protocol* OUTPUT *securely computes $\mathcal{F}_{\textit{output-semi}}$ against a semi-honest adversary who controls $t' = t - k + 1$ parties.*

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

If the Client is corrupted, $\mathcal{S}$ receives the shares of corrupted parties from $\mathcal{F}_{\text{output-semi}}$. $\mathcal{S}$ also receives the secrets $\boldsymbol{x}$ from $\mathcal{F}_{\text{output-semi}}$. Then $\mathcal{S}$ recovers the whole sharing $[\boldsymbol{x}]$ and sends the shares of $[\boldsymbol{x}]$ of honest parties to the Client. Note that a degree-t packed Shamir sharing is determined by the shares of corrupted parties and the secrets, which are $t' + k = t + 1$ shares and secrets. Therefore, the shares of $[\boldsymbol{x}]$ computed by $\mathcal{S}$ are identical to the shares of $[\boldsymbol{x}]$ held by honest parties.

If the Client is honest, $\mathcal{S}$ does not need to do anything. In the real world, the Client is able to recover $\boldsymbol{x}$ from the shares received from all parties. In the ideal world, $\mathcal{F}_{\text{output-semi}}$ can recover the whole sharing from the shares of $[\boldsymbol{x}]$ held by honest parties. Therefore, the Client will receive $\boldsymbol{x}$ from $\mathcal{F}_{\text{output-semi}}$. Thus, the output of the Client is identical in both worlds. □

Evaluating Addition Gates and Multiplication Gates. In the following, we use $[\boldsymbol{x}], [\boldsymbol{y}]$ to denote the input degree-t packed Shamir sharings.

For an addition gate, all parties want to compute $[\boldsymbol{x} + \boldsymbol{y}]$. Note that this can be done by computing $[\boldsymbol{x} + \boldsymbol{y}] := [\boldsymbol{x}] + [\boldsymbol{y}]$, i.e., each party locally adds its shares of $[\boldsymbol{x}], [\boldsymbol{y}]$. The correctness follows from the property that packed Shamir sharing is linearly homomorphic.

Recall that $*$ denotes the coordinate-wise multiplication. For a multiplication gate, all parties want to compute a degree-t packed Shamir sharing of $\boldsymbol{z} := \boldsymbol{x} * \boldsymbol{y}$. We summarize the functionality $\mathcal{F}_{\text{mult-semi}}$ in Functionality 7.

Functionality 7: $\mathcal{F}_{\text{mult-semi}}$

1. Suppose $[\boldsymbol{x}], [\boldsymbol{y}]$ are the input degree-t packed Shamir sharings. $\mathcal{F}_{\text{mult-semi}}$ receives the shares of $[\boldsymbol{x}], [\boldsymbol{y}]$ from honest parties.
2. $\mathcal{F}_{\text{mult-semi}}$ recovers the whole sharings $[\boldsymbol{x}], [\boldsymbol{y}]$ and reconstructs the secrets $\boldsymbol{x}, \boldsymbol{y}$. $\mathcal{F}_{\text{mult-semi}}$ computes $\boldsymbol{z} := \boldsymbol{x} * \boldsymbol{y}$.
3. $\mathcal{F}_{\text{mult-semi}}$ receives from the adversary a set of shares $\{s_i\}_{i \in \mathcal{C}}$. $\mathcal{F}_{\text{mult-semi}}$ samples a random degree-t packed Shamir sharing $[\boldsymbol{z}]$ such that for all $P_i \in \mathcal{C}$, the i-th share of $[\boldsymbol{z}]$ is s_i.
4. $\mathcal{F}_{\text{mult-semi}}$ distributes the shares of $[\boldsymbol{z}]$ to honest parties.

A multiplication gate can be evaluated by a natural extension of the DN multiplication protocol in [DN07]. The main observation is that all parties can locally compute a degree-$2t$ packed Shamir sharing $\langle \boldsymbol{z} \rangle = \langle \boldsymbol{x} * \boldsymbol{y} \rangle = [\boldsymbol{x}] \cdot [\boldsymbol{y}]$. The only task is to reduce the degree of $\langle \boldsymbol{z} \rangle$. Following the approach in [DN07], this can be achieved by preparing a pair of two random sharings $([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$ of the same secrets $\boldsymbol{r}$. All parties do the following steps:

- All parties compute $\langle \boldsymbol{e} \rangle := [\boldsymbol{x}] \cdot [\boldsymbol{y}] + \langle \boldsymbol{r} \rangle$.
- All parties send their shares of $\langle \boldsymbol{e} \rangle$ to the first party P_1.
- P_1 reconstructs the secrets $\boldsymbol{e}$, generates a random degree-t packed Shamir sharing $[\boldsymbol{e}]$, and distributes the shares to all other parties.
- All parties compute $[\boldsymbol{z}] := [\boldsymbol{e}] - [\boldsymbol{r}]$.

The correctness is straightforward. As for secrecy, the degree-$2t$ packed Shamir sharing $\langle \boldsymbol{r} \rangle$ is used as a random mask to protect the multiplication result $\boldsymbol{x} * \boldsymbol{y}$ in the first step. Therefore, P_1 only receives a random degree-$2t$ packed Shamir sharing from all parties.

The preparation of a pair of two random sharings $([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$ can be done by using $\mathcal{F}_{\text{rand}}$ as described in Appendix B.1. The communication complexity of generating m pairs of random sharings in the form of $([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$ is $O(m \cdot n + n^3 \cdot \kappa)$ elements in $\mathbb{F}$. We describe the multiplication protocol MULT in Protocol 8. The communication complexity of m invocations of MULT is $O(m \cdot n + n^3 \cdot \kappa)$ field elements.

Protocol 8: MULT

1. Suppose $[\boldsymbol{x}], [\boldsymbol{y}]$ denote the input packed Shamir sharings of the multiplication gate.
2. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare a pair of random sharings $([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$.
3. All parties locally compute $\langle \boldsymbol{e} \rangle := [\boldsymbol{x}] \cdot [\boldsymbol{y}] + \langle \boldsymbol{r} \rangle$.
4. All parties send their shares of $\langle \boldsymbol{e} \rangle$ to the first party P_1.
5. P_1 reconstructs the secrets $\boldsymbol{e}$, generates a random degree-t packed Shamir sharing $[\boldsymbol{e}]$, and distributes the shares to other parties.
6. All parties locally compute $[\boldsymbol{z}] := [\boldsymbol{e}] - [\boldsymbol{r}]$.

Lemma 3. *Protocol* MULT *securely computes $\mathcal{F}_{\textit{mult-semi}}$ in the $\mathcal{F}_{\textit{rand}}$-hybrid model against a semi-honest adversary who controls $t' = t - k + 1$ parties.*

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

We first prove that, given the shares of $[\boldsymbol{r}], \langle \boldsymbol{r} \rangle$ held by corrupted parties, the shares of $\langle \boldsymbol{r} \rangle$ held by honest parties are uniform. First note that the number of corrupted parties is $|\mathcal{C}| = t - k + 1$, which is a privacy set to both degree-t packed Shamir sharing scheme and degree-$2t$ packed Shamir sharing scheme. I.e., the shares of a degree-t packed Shamir sharing (or a degree-$2t$ packed Shamir sharing) held by corrupted parties are independent of the secret. Recall that $\mathcal{F}_{\text{rand}}$ receives a set of shares $\{s_i\}_{i \in \mathcal{C}}$ from the adversary. Here each share s_i is a pair of elements $(s_i^{(0)}, s_i^{(1)})$ in $\mathbb{F}$. When $\mathcal{F}_{\text{rand}}$ prepares a pair of random sharings $([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$ given the shares held by corrupted parties $\{(s_i^{(0)}, s_i^{(1)})\}_{i \in \mathcal{C}}$, $\mathcal{F}_{\text{rand}}$ can do the following steps:

1. $\mathcal{F}_{\text{rand}}$ first prepares a degree-$2t$ packed Shamir sharing $\langle \boldsymbol{r} \rangle$. This is done by choosing a random element in $\mathbb{F}$ for each honest party. Then using the shares $\{s_i^{(1)}\}_{i \in \mathcal{C}}$, $\mathcal{F}_{\text{rand}}$ reconstructs the whole sharing $\langle \boldsymbol{r} \rangle$ and the secrets $\boldsymbol{r}$.
2. $\mathcal{F}_{\text{rand}}$ then prepares a degree-t packed Shamir sharing $[\boldsymbol{r}]$. Based on the shares $\{s_i^{(0)}\}_{i \in \mathcal{C}}$ and the secrets $\boldsymbol{r}$, $\mathcal{F}_{\text{rand}}$ reconstructs the whole sharing $[\boldsymbol{r}]$.

Note that the above steps output a pair of random sharings $([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$ given the shares $\{(s_i^{(0)}, s_i^{(1)})\}_{i \in \mathcal{C}}$. In particular, the shares of $\langle \boldsymbol{r} \rangle$ held by honest parties are uniform given the shares $\{(s_i^{(0)}, s_i^{(1)})\}_{i \in \mathcal{C}}$.

Now we describe the construction of the simulator $\mathcal{S}$. In Step 2, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ and receives the shares $\{(s_i^{(0)}, s_i^{(1)})\}_{i \in \mathcal{C}}$ from the adversary. In Step 3, $\mathcal{S}$ generates a random element in $\mathbb{F}$ for each honest party as its share of $\langle \boldsymbol{e} \rangle$. Since the shares of $\langle \boldsymbol{r} \rangle$ of honest parties in the real world are uniform, the distribution of the shares of $\langle \boldsymbol{e} \rangle$ of honest parties generated by $\mathcal{S}$ is identical to that in the real world. $\mathcal{S}$ honestly follows Step 4 and Step 5. At the end of Step 5, $\mathcal{S}$ learns the shares of $[\boldsymbol{e}]$ held by honest parties. In Step 6, $\mathcal{S}$ computes the shares of $[\boldsymbol{z}]$ held by corrupted parties. This can be computed by using the shares of $[\boldsymbol{e}]$ and $[\boldsymbol{r}]$ held by corrupted parties, where

- the shares of $[\boldsymbol{e}]$ held by corrupted parties can be computed by the shares of honest parties $\mathcal{S}$ learnt in Step 5,
- and the shares of $[\boldsymbol{r}]$ held by corrupted parties are received when $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ in Step 2.

Then $\mathcal{S}$ sends the shares of $[\boldsymbol{z}]$ held by corrupted parties to $\mathcal{F}_{\text{mult-semi}}$.

It is clear that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. As for the output, note that the whole sharing $[\boldsymbol{z}]$ is determined by the secrets $\boldsymbol{z}$ and the shares held by corrupted parties, which are identical in both worlds. Therefore, the output of honest parties is identical in both worlds. $\square$

4.2 Performing an Arbitrary Permutation on the Secrets of a Single Sharing

During the computation, we may encounter the scenario that the order of the secrets is not what we want. For example, when using $\mathcal{F}_{\text{mult-semi}}$ to evaluate k multiplication gates, for all $i \in [k]$, we need the i-th secrets of both input degree-t packed Shamir sharings to be the input of the i-th multiplication gate. If the secrets of either sharing are not in the correct order, we cannot get the correct multiplication result.

To solve it, we need a functionality which allows us to perform an arbitrary permutation on the secrets of a single sharing. Let $p(\cdot)$ be a permutation over $\{1, 2, \ldots, k\}$. Recall that each permutation $p(\cdot)$ maps to a permutation matrix $\boldsymbol{M}_p \in \{0,1\}^{k\times k}$ where $(\boldsymbol{M}_p)_{i,j} = 1$ iff $p(i) = j$. To permute a vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_k)$ to $\tilde{\boldsymbol{x}} = (x_{p(1)}, x_{p(2)}, \ldots, x_{p(k)})$, it is equivalent to computing $\tilde{\boldsymbol{x}} = \boldsymbol{M}_p \cdot \boldsymbol{x}$. We model the functionality $\mathcal{F}_{\text{permute-semi}}$ in Functionality 9.

Functionality 9: $\mathcal{F}_{\text{permute-semi}}$

1. $\mathcal{F}_{\text{permute-semi}}$ receives a permutation p and the shares of a degree-t packed Shamir sharing $[\boldsymbol{x}]$ from honest parties.
2. $\mathcal{F}_{\text{permute-semi}}$ reconstructs the secrets $\boldsymbol{x}$ from the shares of honest parties, and computes $\tilde{\boldsymbol{x}} = \boldsymbol{M}_p \cdot \boldsymbol{x}$.
3. $\mathcal{F}_{\text{permute-semi}}$ receives from the adversary a set of shares $\{s_i\}_{i\in\mathcal{C}}$. $\mathcal{F}_{\text{permute-semi}}$ samples a random degree-t packed Shamir sharing $[\tilde{\boldsymbol{x}}]$ such that for all $P_i \in \mathcal{C}$, the i-th share of $[\tilde{\boldsymbol{x}}]$ is s_i.
4. $\mathcal{F}_{\text{permute-semi}}$ distributes the shares of $[\tilde{\boldsymbol{x}}]$ to honest parties.

Following the techniques in [DIK10], $\mathcal{F}_{\text{permute-semi}}$ can be realized as follows:

1. All parties prepare two random degree-t packed Shamir sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$, where $\tilde{\boldsymbol{r}} = \boldsymbol{M}_p \cdot \boldsymbol{r}$ and $p(\cdot)$ is the permutation we want to perform.
2. All parties locally compute $[\boldsymbol{e}] := [\boldsymbol{x}] + [\boldsymbol{r}]$.
3. All parties send their shares of $[\boldsymbol{e}]$ to the first party P_1.
4. P_1 reconstructs the secrets $\boldsymbol{e}$ and computes $\tilde{\boldsymbol{e}} = \boldsymbol{M}_p \cdot \boldsymbol{e}$. P_1 generates a random degree-t packed Shamir sharing $[\tilde{\boldsymbol{e}}]$ and distributes the shares to other parties.
5. All parties locally compute $[\tilde{\boldsymbol{x}}] := [\tilde{\boldsymbol{e}}] - [\tilde{\boldsymbol{r}}]$.

To see the correctness, note that in the second step we have $\boldsymbol{e} = \boldsymbol{x} + \boldsymbol{r}$. Therefore,

$$\tilde{\boldsymbol{x}} = \boldsymbol{M}_p \cdot \boldsymbol{x} = \boldsymbol{M}_p \cdot (\boldsymbol{e} - \boldsymbol{r}) = \boldsymbol{M}_p \cdot \boldsymbol{e} - \boldsymbol{M}_p \cdot \boldsymbol{r} = \tilde{\boldsymbol{e}} - \tilde{\boldsymbol{r}}.$$

As noted in [DIK10], the main issue of this approach is how to *efficiently* prepare a pair of random sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$. Indeed, such a pair of random sharings can be prepared using $\mathcal{F}_{\text{rand}}$ with a suitable $\mathbb{F}$-linear secret sharing scheme. However, for different permutations, the $\mathbb{F}$-linear secret sharing schemes used in $\mathcal{F}_{\text{rand}}$ are also different. Although the amortized communication complexity of the implementation of $\mathcal{F}_{\text{rand}}$ is $O(n)$ field elements, it has an overhead of $O(n^3 \cdot \kappa)$ field elements which is independent of the number of sharings we want to prepare. In the worst case where each time we need to perform a different permutation, the overhead of preparing each pair of random sharings becomes $O(n^3 \cdot \kappa)$, which eliminates the benefit we gain when using the packed Shamir sharing scheme. In [DIK10], this issue is solved by compiling the circuit such that only $O(\log n)$ different permutations are needed in the computation with the cost of blowing up the circuit size and the circuit depth by a factor of $O(\log |C|)$, where $|C|$ is the circuit size. This approach does not achieve our goal since the amortized communication complexity per gate becomes $O(\log |C| \cdot n/k)$ elements.

Before introducing our idea, we first introduce a useful functionality $\mathcal{F}_{\text{select}}$, which selects secrets from one or more packed Shamir sharings and outputs a single sharing which contains the chosen secrets. Later on, we will use $\mathcal{F}_{\text{select}}$ to solve the above issue of preparing random sharings for permutations.

Selecting Secrets from One or More Packed Shamir Sharings. Concretely, we want to realize the functionality $\mathcal{F}_{\text{select-semi}}$, which takes as input k degree-t packed Shamir sharings $[\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(k)}]$ and a permutation $p(\cdot)$ over $\{1, 2, \ldots, k\}$, and outputs a degree-t packed Shamir sharing $[\boldsymbol{y}]$ such that for all $i \in [k]$, $y_{p(i)} = x^{(i)}_{p(i)}$, where $x^{(i)}_j$ is the j-th value of $\boldsymbol{x}^{(i)}$. Effectively, $\mathcal{F}_{\text{select-semi}}$ chooses the $p(1)$-th secret of $[\boldsymbol{x}^{(1)}]$, the $p(2)$-th secret of $[\boldsymbol{x}^{(2)}]$, ..., the $p(k)$-th secret of $[\boldsymbol{x}^{(k)}]$ and generates a new degree-t packed Shamir sharing $[\boldsymbol{y}]$ which contains the chosen secrets. Note that the positions of the chosen secrets remain the same. Therefore, we require p to be a permutation so that the chosen secrets come from different positions. The description of $\mathcal{F}_{\text{select-semi}}$ appears in Functionality 10.

Functionality 10: $\mathcal{F}_{\text{select-semi}}$

1. $\mathcal{F}_{\text{select-semi}}$ receives from honest parties their shares of k degree-t packed Shamir sharings $[\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(k)}]$. $\mathcal{F}_{\text{select-semi}}$ also receives a permutation p from honest parties.
2. $\mathcal{F}_{\text{select-semi}}$ reconstructs $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(k)}$. Then $\mathcal{F}_{\text{select-semi}}$ sets $\boldsymbol{y} = (y_1, y_2, \ldots, y_k)$ such that for all $i \in [k]$, $y_{p(i)} = x^{(i)}_{p(i)}$, where $x^{(i)}_j$ is the j-th value of $\boldsymbol{x}^{(i)}$.
3. $\mathcal{F}_{\text{select-semi}}$ receives from the adversary a set of shares $\{s_i\}_{i \in \mathcal{C}}$. $\mathcal{F}_{\text{select-semi}}$ samples a random degree-t packed Shamir sharing $[\boldsymbol{y}]$ such that for all $P_i \in \mathcal{C}$, the i-th share of $[\boldsymbol{y}]$ is s_i.
4. $\mathcal{F}_{\text{select-semi}}$ distributes the shares of $[\boldsymbol{y}]$ to honest parties.

For all $i \in [k]$, let $\boldsymbol{e}_i \in \{0,1\}^k$ denote the vector where the i-th entry is 1 and for all $j \neq i$, the j-th entry is 0. Recall that in Section 3.2 we show how to transform a constant vector to a degree-t packed Shamir sharing. Let $[\boldsymbol{e}_i]$ denote the degree-t packed Shamir sharing of $\boldsymbol{e}_i$.

To realize $\mathcal{F}_{\text{select-semi}}$, note that $[\boldsymbol{e}_{p(i)}] \cdot [\boldsymbol{x}^{(i)}]$ is a degree-$2t$ packed Shamir sharing of $\boldsymbol{e}_{p(i)} * \boldsymbol{x}^{(i)}$. Also note that $\boldsymbol{y} = \sum_{i=1}^{k} \boldsymbol{e}_{p(i)} * \boldsymbol{x}^{(i)}$. Therefore, all parties can locally compute $\langle \boldsymbol{y} \rangle = \sum_{i=1}^{k} [\boldsymbol{e}_{p(i)}] \cdot [\boldsymbol{x}^{(i)}]$. And the only task is to reduce the degree of $\langle \boldsymbol{y} \rangle$. Note that this can be achieved by the same technique as Mult. The description of the protocol Select appears in Protocol 11. The communication complexity of m invocations of Select is $O(m \cdot n + n^3 \cdot \kappa)$ field elements.

Lemma 4. *Protocol* Select *securely computes $\mathcal{F}_{\textit{select-semi}}$ in the $\mathcal{F}_{\textit{rand}}$-hybrid model against a semi-honest adversary who controls $t' = t - k + 1$ parties.*

This lemma can be proved in the same way as that for Lemma 3. Therefore, for simplicity, we omit the details.

Using $\mathcal{F}_{\text{select-semi}}$ to Generate Random Sharings for Permuting Secrets. For all $i, j \in \{1, 2, \ldots, k\}$, we say a pair of degree-t packed Shamir sharings $([\boldsymbol{x}], [\boldsymbol{y}])$ contains an (i, j)-component if the secrets of these two sharings satisfy that $x_i = y_j$.

Protocol 11: Select

1. Let $[\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(k)}]$ denote the k input packed Shamir sharings and $p(\cdot)$ denote the permutation. The goal is to generate a degree-t packed Shamir sharing $[\boldsymbol{y}]$ such that for all $i \in [k]$, $y_{p(i)} = x^{(i)}_{p(i)}$. Recall that $\boldsymbol{e}_i \in \{0,1\}^k$ denote the vector where the i-th entry is 1 and for all $j \neq i$, the j-th entry is 0. For all $i \in [k]$, all parties agree on the whole sharing $[\boldsymbol{e}_i]$ based on the transformation in Section 3.2.

2. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare a pair of random sharings $([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$.

3. All parties locally compute $\langle \boldsymbol{e} \rangle := \sum_{i=1}^{k} [\boldsymbol{e}_{p(i)}] \cdot [\boldsymbol{x}^{(i)}] + \langle \boldsymbol{r} \rangle$.

4. All parties send their shares of $\langle \boldsymbol{e} \rangle$ to the first party P_1.

5. P_1 reconstructs the secrets $\boldsymbol{e}$, generates a random degree-t packed Shamir sharing $[\boldsymbol{e}]$, and distributes the shares to other parties.

6. All parties locally compute $[\boldsymbol{y}] := [\boldsymbol{e}] - [\boldsymbol{r}]$.

To perform a permutation $p(\cdot)$, we need to prepare two random degree-t packed Shamir sharings $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}])$. We can view $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}])$ as a composition of a $(1, p(1))$-component, a $(2, p(2))$-component, $\ldots$, and a $(k, p(k))$-component.

Let m denote the number of permutations we want to prepare random sharings for. These permutations are denoted by $p_1(\cdot), p_2(\cdot), \ldots, p_m(\cdot)$. Our idea is as follows:

1. We first find m permutations $q_1(\cdot), q_2(\cdot), \ldots, q_m(\cdot)$ such that:
   - For all $i, j \in \{1, 2, \ldots, k\}$, the number of permutations $p \in \{p_1, p_2, \ldots, p_m\}$ which satisfies that $p(i) = j$ is equal to the number of permutations $q \in \{q_1, q_2, \ldots, q_m\}$ which satisfies that $q(i) = j$.
2. All parties prepare random sharings for permutations $q_1, q_2, \ldots, q_m$.
3. From $i = 1$ to m, a pair of random sharings for the permutation p_i is prepared by using $\mathcal{F}_{\text{select-semi}}$ to choose the first unused $(j, p_i(j))$-component from the random sharings for $q_1, q_2, \ldots, q_m$ for all $j \in [k]$.

We refer the readers to Section 2.2 for a more detailed explanation.

The major benefit of this approach is that *we can limit the number of different permutations in* $\{q_1, q_2, \ldots, q_m\}$ as we show below. More concretely, we will prove the following theorem:

Theorem 2. *Let $m, k \geq 1$ be integers. For all m permutations $p_1, p_2, \ldots, p_m$ over $\{1, 2, \ldots, k\}$, there exists m permutations $q_1, q_2, \ldots, q_m$ over $\{1, 2, \ldots, k\}$ such that:*

- *For all $i, j \in \{1, 2, \ldots, k\}$, the number of permutations $p \in \{p_1, p_2, \ldots, p_m\}$ such that $p(i) = j$ is the same as the number of permutations $q \in \{q_1, q_2, \ldots, q_m\}$ such that $q(i) = j$.*
- *$q_1, q_2, \ldots, q_m$ contain at most k^2 different permutations.*

Moreover, $q_1, q_2, \ldots, q_m$ can be found within polynomial time given $p_1, p_2, \ldots, p_m$.

Recall that the issue of using $\mathcal{F}_{\text{rand}}$ to prepare random sharings for $p_1, p_2, \ldots, p_m$ is that there will be an overhead of $O(n^3 \cdot \kappa)$ elements per different permutation in $p_1, p_2, \ldots, p_m$. Since there are $k!$ different permutations over $\{1, 2, \ldots, k\}$, $p_1, p_2, \ldots, p_m$ can be m different permutations. In the worst case, this overhead can be as large as $O(n^3 \cdot \kappa \cdot m)$, which eliminates the benefit of using the packed Shamir sharing scheme. Relying on $\mathcal{F}_{\text{select-semi}}$, we only need to prepare random sharings for permutations $q_1, \ldots, q_m$, which contain at most $k^2 \leq n^2$ different permutations. In this way, the overhead is upper-bounded by $O(n^5 \cdot \kappa)$, which is independent of the number of permutations and the circuit size.

Proof of Theorem 2. Recall that each permutation $p(\cdot)$ over $\{1, 2, \ldots, k\}$ maps to a permutation matrix $\boldsymbol{M}_p \in \{0,1\}^{k \times k}$ such that for all $i, j \in \{1, 2, \ldots, k\}$, $(\boldsymbol{M}_p)_{i,j} = 1$ iff $p(i) = j$.

Let $\boldsymbol{M}_{\text{demand}} := \sum_{i=1}^{m} \boldsymbol{M}_{p_i}$, where $p_1, p_2, \ldots, p_m$ are the given permutations. Then $(\boldsymbol{M}_{\text{demand}})_{i,j}$ is the number of permutations $p \in \{p_1, p_2, \ldots, p_m\}$ such that $p(i) = j$. Therefore, our goal is to find m permutations $q_1, q_2, \ldots, q_m$, which contain at most k^2 different permutations, such that $\sum_{i=1}^{m} \boldsymbol{M}_{q_i} = \boldsymbol{M}_{\text{demand}}$.

Recall that a matrix $\boldsymbol{M} \in \mathbb{N}^{k \times k}$ is a balanced matrix if for each row and each column, the summation of all the entries is the same. Since each permutation matrix is a balanced matrix, $\boldsymbol{M}_{\text{demand}}$ is also a balanced matrix.

Consider the following process to find the permutations $q_1, q_2, \ldots, q_m$.

1. Initially set the list of permutation Q to be empty. Let $\boldsymbol{N} = \boldsymbol{M}_{\text{demand}}$.
2. While $\boldsymbol{N}$ is non-zero, run the following steps:
   (a) Find a permutation matrix $\boldsymbol{M}$ such that for all $i, j \in \{1, 2, \ldots, k\}$, $\boldsymbol{N}_{i,j} \geq \boldsymbol{M}_{i,j}$.
   (b) Let q' denote the permutation which corresponds to the permutation matrix $\boldsymbol{M}$. Let
   $$n' = \min\{\boldsymbol{N}_{1,q'(1)}, \boldsymbol{N}_{2,q'(2)}, \ldots, \boldsymbol{N}_{k,q'(k)}\}.$$
   Insert n' times of q' into the list Q.
   (c) Compute $\boldsymbol{N} := \boldsymbol{N} - n' \cdot \boldsymbol{M}$. Note that $\boldsymbol{N}$ is still a balanced matrix.
3. Output the permutations in Q.

We first show that the above process will finally terminate. In the beginning, $\boldsymbol{N} = \boldsymbol{M}_{\text{demand}}$ is a balanced matrix. $\boldsymbol{N}$ represents the summation of the permutation matrices we still need to generate. We show that this property is maintained in Step 2. In Step 2.(a), according to Corollary 2, such a permutation matrix exists and can be found within polynomial time. In Step 2.(b), n' is the largest number of q' we can have. In Step 3, we compute the new matrix $\boldsymbol{N}$ by subtracting n' times of $\boldsymbol{M}$. Note that each entry of the new matrix $\boldsymbol{N}$ is still non-negative. Since the permutation matrix $\boldsymbol{M}$ we subtract is also a balanced matrix, for each row and each column of the new matrix $\boldsymbol{N}$, the summation of all the entries is still the same. Therefore, the new matrix $\boldsymbol{N}$ is a balanced matrix. Note that the summation of all the entries in $\boldsymbol{N}$ decreases in each iteration. We conclude that the above process will finally terminate.

Now we show that the number of different permutations in Q is bounded by k^2. It is sufficient to show that Step 2 will terminate within k^2 rounds since in each round we only insert the same type of permutations into Q. To this end, we count the number of 0-entries in $\boldsymbol{N}$. Note that in each round, the way we choose n' in Step 2.(b) guarantees that at least one entry of $\boldsymbol{N}$ will become 0 after subtracting n' times of $\boldsymbol{M}$. Therefore, the number of 0-entries in $\boldsymbol{N}$ increases at least by 1 in each round. Since there are k^2 entries in $\boldsymbol{N}$, Step 2 will terminate within k^2 rounds. □

Preparing Random Sharings for Different Permutations. We are ready to introduce the functionality and its implementation for preparing random sharings for different permutations. The functionality $\mathcal{F}_{\text{rand-perm-semi}}$ appears in Functionality 12.

For a fixed permutation $p(\cdot)$ over $\{1, 2, \ldots, k\}$, we show how to use $\mathcal{F}_{\text{rand}}$ to prepare a pair of random sharings $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}])$ in Appendix B.2. The communication complexity of preparing m pairs of random sharings in the form of $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}])$ for a fixed permutation $p(\cdot)$ is $O(m \cdot n + n^3 \cdot \kappa)$ elements in $\mathbb{F}$. We describe the protocol for $\mathcal{F}_{\text{rand-perm-semi}}$ in Protocol 13. The communication complexity of using Rand-Perm to prepare random sharings for m permutations is $O(m \cdot n + n^5 \cdot \kappa)$ field elements.

Functionality 12: $\mathcal{F}_{\text{rand-perm-semi}}$

1. $\mathcal{F}_{\text{rand-perm-semi}}$ receives from honest parties m permutations $p_1, p_2, \ldots, p_m$ over $\{1, 2, \ldots, k\}$.
2. For all $i \in [m]$, $\mathcal{F}_{\text{rand-perm-semi}}$ receives from the adversary a set of shares $\{(u_j^{(i)}, v_j^{(i)})\}_{j \in \mathcal{C}}$. $\mathcal{F}_{\text{rand-perm-semi}}$ samples a random vector $\boldsymbol{r}^{(i)} \in \mathbb{F}^k$ and samples two degree-t packed Shamir sharings $([\boldsymbol{r}^{(i)}], [\boldsymbol{M}_{p_i} \cdot \boldsymbol{r}^{(i)}])$ such that for all $P_j \in \mathcal{C}$, the j-th share of $([\boldsymbol{r}^{(i)}], [\boldsymbol{M}_{p_i} \cdot \boldsymbol{r}^{(i)}])$ is $(u_j^{(i)}, v_j^{(i)})$.
3. For all $i \in [m]$, $\mathcal{F}_{\text{rand-perm-semi}}$ distributes the shares of $([\boldsymbol{r}^{(i)}], [\boldsymbol{M}_{p_i} \cdot \boldsymbol{r}^{(i)}])$ to honest parties.

Lemma 5. *Protocol* Rand-Perm *securely computes $\mathcal{F}_{\text{rand-perm-semi}}$ in the $(\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{select-semi}})$-hybrid model against a semi-honest adversary who controls $t' = t - k + 1$ parties.*

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

In Step 1 and Step 2, $\mathcal{S}$ honestly follows the protocol and computes the permutations $q_1, q_2, \ldots, q_m$. In Step 3, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ when generating the random sharings $([\boldsymbol{r}^{(i)}], [\boldsymbol{M}_{q_i} \cdot \boldsymbol{r}^{(i)}])$ for each permutation q_i. Since the number of corrupted parties is $t - k + 1$, by the property of the degree-t packed Shamir sharing scheme, the secrets $\boldsymbol{r}^{(i)}$ are uniform and independent of the shares held by corrupted parties. In Step 4, $\mathcal{S}$ honestly constructs the lists. In Step 5, $\mathcal{S}$ emulates $\mathcal{F}_{\text{select-semi}}$ and receives the shares of corrupted parties for all $([\boldsymbol{v}^{(\ell)}], [\tilde{\boldsymbol{v}}^{(\ell)}])$. Then, $\mathcal{S}$ sends these shares to $\mathcal{F}_{\text{rand-perm-semi}}$.

Note that through the protocol, corrupted parties do not receive any messages from honest parties. Therefore, it is sufficient to argue that the distribution of the output in both worlds are identical. First, for all $\ell \in \{1, 2, \ldots, m\}$, the shares of $[\boldsymbol{v}^{(\ell)}]$ and $[\tilde{\boldsymbol{v}}^{(\ell)}]$ held by corrupted parties are chosen by themselves in both worlds. For the secrets $\boldsymbol{v}^{(\ell)}$ and $\tilde{\boldsymbol{v}}^{(\ell)}$, in the ideal world, $\boldsymbol{v}^{(\ell)}$ is a uniform vector in $\mathbb{F}^k$ and $\tilde{\boldsymbol{v}}^{(\ell)} = \boldsymbol{M}_{p_\ell} \cdot \boldsymbol{v}^{(\ell)}$. In the real world, there is a one-to-one map from $\{\boldsymbol{r}^{(\ell)}\}_{\ell \in [m]}$ to $\{\boldsymbol{v}^{(\ell)}\}_{\ell \in [m]}$. Therefore each vector $\boldsymbol{v}^{(\ell)}$ is uniform. As for $\tilde{\boldsymbol{v}}^{(\ell)}$, note that in Step 5, Case 2.2, we have $v_i^{(\ell)} = r_i^{(\ell_i)} = (\boldsymbol{M}_{q_{\ell_i}} \cdot \boldsymbol{r}^{(\ell_i)})_{q_{\ell_i}(i)} = (\boldsymbol{M}_{q_{\ell_i}} \cdot \boldsymbol{r}^{(\ell_i)})_{p_\ell(i)} = \tilde{v}_{p_\ell(i)}^{(\ell)}$ for all $i \in [k]$. Therefore $\tilde{\boldsymbol{v}}^{(\ell)} = \boldsymbol{M}_{p_\ell} \cdot \boldsymbol{v}^{(\ell)}$. Thus, the distribution of the output in both worlds are identical. □

Realizing $\mathcal{F}_{\text{permute-semi}}$. Now we are ready to present the protocol for $\mathcal{F}_{\text{permute-semi}}$. The protocol Permute uses $\mathcal{F}_{\text{rand-perm-semi}}$ to prepare the random sharings for the permutation we want to perform and then follows the techniques in [DIK10]. In Permute, we will prepare a random degree-$2t$ packed Shamir sharing of $\boldsymbol{0} \in \mathbb{F}^k$, which is used as a random mask for the shares of honest parties (see the proof of Lemma 6). This is not needed for semi-honest security but will be helpful when we consider a fully malicious adversary at a later point.

We show how to use $\mathcal{F}_{\text{rand}}$ to prepare a random degree-$2t$ packed Shamir sharing of $\boldsymbol{0} \in \mathbb{F}^k$ in Appendix B.3. The communication complexity of preparing m random degree-$2t$ packed Shamir sharings of $\boldsymbol{0}$ is $O(m \cdot n + n^3 \cdot \kappa)$ elements in $\mathbb{F}$. The description of Permute appears in Protocol 14. The communication complexity of m invocations of Permute is $O(m \cdot n + n^5 \cdot \kappa)$ field elements.

Lemma 6. *Protocol* Permute *securely computes $\mathcal{F}_{\text{permute-semi}}$ in the $(\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{rand-perm-semi}})$-hybrid model against a semi-honest adversary who controls $t' = t - k + 1$ parties.*

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

Let $[\boldsymbol{r}]$ be a random degree-t packed Shamir sharing, and $\langle \boldsymbol{0} \rangle$ be a random degree-$2t$ packed Shamir sharing of $\boldsymbol{0} \in \mathbb{F}^k$. We show that, given the shares of $[\boldsymbol{r}]$ and $\langle \boldsymbol{0} \rangle$ held by corrupted parties, the shares of $\langle \boldsymbol{r} \rangle := [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle$ held by honest parties are uniform. Note that there exists a one-to-one map from $([\boldsymbol{r}], \langle \boldsymbol{0} \rangle)$ to

Protocol 13: Rand-Perm

1. Let $p_1, p_2, \ldots, p_m$ be the permutations over $\{1, 2, \ldots, k\}$ that all parties want to prepare random sharings for.

2. All parties use a deterministic algorithm that all parties agree on to compute m permutations $q_1, q_2, \ldots, q_m$ such that

   - For all $i, j \in \{1, 2, \ldots, k\}$, the number of permutations $p \in \{p_1, p_2, \ldots, p_m\}$ such that $p(i) = j$ is the same as the number of permutations $q \in \{q_1, q_2, \ldots, q_m\}$ such that $q(i) = j$.
   - $q_1, q_2, \ldots, q_m$ contain at most k^2 different permutations.

   The existence of such an algorithm is guaranteed by Theorem 2.

3. Suppose $q'_1, q'_2, \ldots, q'_{k^2}$ denote the different permutations in $q_1, q_2, \ldots, q_m$. For all $i \in \{1, 2, \ldots, k^2\}$, let n'_i denote the number of times that q'_i appears in $q_1, q_2, \ldots, q_m$. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare n'_i pairs of random sharings in the form $([\boldsymbol{r}], [\boldsymbol{M}_{q'_i} \cdot \boldsymbol{r}])$ for all $i \in \{1, 2, \ldots, k^2\}$. Note that we have prepared a pair of random sharings for each permutation q_i for all $i \in [m]$. Let $([\boldsymbol{r}^{(i)}], [\boldsymbol{M}_{q_i} \cdot \boldsymbol{r}^{(i)}])$ denote the random sharings for the permutation q_i.

4. For all $i, j \in \{1, 2, \ldots, k\}$, all parties initiate an empty list $L_{i,j}$. From $\ell = 1$ to m, for all $i, j \in \{1, 2, \ldots, k\}$, if $([\boldsymbol{r}^{(\ell)}], [\boldsymbol{M}_{q_\ell} \cdot \boldsymbol{r}^{(\ell)}])$ contains an (i, j)-component, all parties insert $([\boldsymbol{r}^{(\ell)}], [\boldsymbol{M}_{q_\ell} \cdot \boldsymbol{r}^{(\ell)}])$ into the list $L_{i,j}$.

5. From $\ell = 1$ to m, all parties prepare a pair of random sharings for p_ℓ as follows:

   - From $i = 1$ to k, let $([\boldsymbol{r}^{(\ell_i)}], [\boldsymbol{M}_{q_{\ell_i}} \cdot \boldsymbol{r}^{(\ell_i)}])$ denote the first pair of sharings in the list $L_{i,p_\ell(i)}$, and then remove it from $L_{i,p_\ell(i)}$. Note that $([\boldsymbol{r}^{(\ell_i)}], [\boldsymbol{M}_{q_{\ell_i}} \cdot \boldsymbol{r}^{(\ell_i)}])$ contains an $(i, p_\ell(i))$-component, which is not used when preparing random sharings for $p_1, p_2, \ldots, p_{\ell-1}$.
   - Let I denote the identity permutation over $\{1, 2, \ldots, k\}$.
     - All parties invoke $\mathcal{F}_{\text{select-semi}}$ with
       $$[\boldsymbol{r}^{(\ell_1)}], [\boldsymbol{r}^{(\ell_2)}], \ldots, [\boldsymbol{r}^{(\ell_k)}]$$
       and the permutation I. The output is denoted by $[\boldsymbol{v}^{(\ell)}]$.
     - All parties invoke $\mathcal{F}_{\text{select-semi}}$ with
       $$[\boldsymbol{M}_{q_{\ell_1}} \cdot \boldsymbol{r}^{(\ell_1)}], [\boldsymbol{M}_{q_{\ell_2}} \cdot \boldsymbol{r}^{(\ell_2)}], \ldots, [\boldsymbol{M}_{q_{\ell_k}} \cdot \boldsymbol{r}^{(\ell_k)}]$$
       and the permutation p_ℓ. The output is denoted by $[\tilde{\boldsymbol{v}}^{(\ell)}]$. Note that for all $i \in [k]$, $v_i^{(\ell)} = r_i^{(\ell_i)} = (\boldsymbol{M}_{q_{\ell_i}} \cdot \boldsymbol{r}^{(\ell_i)})_{q_{\ell_i}(i)} = (\boldsymbol{M}_{q_{\ell_i}} \cdot \boldsymbol{r}^{(\ell_i)})_{p_\ell(i)} = \tilde{v}^{(\ell)}_{p_\ell(i)}$.

6. All parties take $([\boldsymbol{v}^{(1)}], [\tilde{\boldsymbol{v}}^{(1)}]), ([\boldsymbol{v}^{(2)}], [\tilde{\boldsymbol{v}}^{(2)}]), \ldots, ([\boldsymbol{v}^{(m)}], [\tilde{\boldsymbol{v}}^{(m)}])$ as output.

$([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$. Recall that in Lemma 3, we have shown that, given the shares of $[\boldsymbol{r}]$ and $\langle \boldsymbol{r} \rangle$ of corrupted parties, the shares of of $\langle \boldsymbol{r} \rangle$ held by honest parties are uniform. Since the shares of $\langle \boldsymbol{0} \rangle$ held by corrupted parties can be computed from the shares of $\langle \boldsymbol{r} \rangle$ and $[\boldsymbol{r}]$ held by corrupted parties, the statement holds.

Now we describe the construction of $\mathcal{S}$. In Step 2, $\mathcal{S}$ emulates the $\mathcal{F}_{\text{rand-perm-semi}}$ and $\mathcal{F}_{\text{rand}}$, and receives the shares of $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}])$ and $\langle \boldsymbol{0} \rangle$ held by corrupted parties. In Step 3, for each honest party, $\mathcal{S}$ generates a random element in $\mathbb{F}$ as its share of $\langle \boldsymbol{e} \rangle$. As we argued above, the share of $\langle \boldsymbol{r} \rangle := [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle$ held by each

Protocol 14: Permute

1. Let $[\boldsymbol{x}]$ denote the input degree-t packed Shamir sharing and $p(\cdot)$ denote the permutation all parties want to perform on $\boldsymbol{x}$.
2. All parties invoke $\mathcal{F}_{\text{rand-perm-semi}}$ with p to prepare a pair of random sharings $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}])$. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare a random degree-$2t$ packed Shamir sharing $\langle \boldsymbol{0} \rangle$.
3. All parties locally compute $\langle \boldsymbol{e} \rangle := [\boldsymbol{x}] + [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle$.
4. All parties send their shares of $\langle \boldsymbol{e} \rangle$ to the first party P_1.
5. P_1 reconstructs the secrets $\boldsymbol{e}$, and computes $\tilde{\boldsymbol{e}} = \boldsymbol{M}_p \cdot \boldsymbol{e}$. Then P_1 generates a random degree-t packed Shamir sharing $[\tilde{\boldsymbol{e}}]$, and distributes the shares to other parties.
6. All parties locally compute $[\tilde{\boldsymbol{x}}] := [\tilde{\boldsymbol{e}}] - [\boldsymbol{M}_p \cdot \boldsymbol{r}]$.

honest party is uniform. Therefore, the share of $\langle \boldsymbol{e} \rangle$ held by each honest party is also uniform. In Step 4, Step 5 and Step 6, $\mathcal{S}$ honestly follows the protocol. At the end of the protocol, $\mathcal{S}$ computes the shares of $[\tilde{\boldsymbol{x}}]$ held by corrupted parties. This can be computed by using the shares of $[\tilde{\boldsymbol{e}}]$ and $[\boldsymbol{M}_p \cdot \boldsymbol{r}]$ held by corrupted parties, where

- the shares of $[\tilde{\boldsymbol{e}}]$ held by corrupted parties can be computed by the shares of honest parties $\mathcal{S}$ learnt in Step 5,
- and the shares of $[\boldsymbol{M}_p \cdot \boldsymbol{r}]$ held by corrupted parties are received when $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand-perm-semi}}$ in Step 2.

Then $\mathcal{S}$ sends the shares of $[\tilde{\boldsymbol{x}}]$ held by corrupted parties to $\mathcal{F}_{\text{permute-semi}}$.

It is clear that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. As for the output, note that the whole sharing $[\tilde{\boldsymbol{x}}]$ is determined by the secrets $\tilde{\boldsymbol{x}}$ and the shares of corrupted parties. In both worlds, $\tilde{\boldsymbol{x}} = \boldsymbol{M}_p \cdot \boldsymbol{x}$. As for the shares of $[\tilde{\boldsymbol{x}}]$ held by corrupted parties, they are chosen by the adversary in both worlds. Therefore, the output of honest parties is identical in both worlds. □

Remark 2. *We note that the functionality $\mathcal{F}_{\text{rand-perm-semi}}$ needs to know all the permutations before generating the random sharings. In the real execution, all the invocations of* Permute *will first run the second step to submit the permutation to $\mathcal{F}_{\text{rand-perm-semi}}$. This can be done since the permutations that needed to be performed during the computation are determined by the circuit and are independent of the inputs.*

4.3 Obtaining Input Sharings for Multiplication Gates and Addition Gates

So far, we have introduced how to evaluate multiplication gates and addition gates using the packed Shamir sharing scheme. In the case that the secrets of an input sharing are not in the correct order, we have shown how to efficiently perform a permutation to obtain the correct order. During the computation, however, input sharings of multiplication gates and addition gates do not come for free. When evaluating the multiplication gates and addition gates in some layer, the secrets we want to be in a single sharing may be scattered in one or more output sharings from the previous layers. This requires us to collect the secrets from those sharings and generate a single sharing for these secrets efficiently.

Our starting point is the functionality $\mathcal{F}_{\text{select-semi}}$. Recall that $\mathcal{F}_{\text{select-semi}}$ allows us to select secrets from one or more sharings and generate a new sharing for the chosen secrets if the secrets we select are *in different positions.* To use $\mathcal{F}_{\text{select-semi}}$, we consider what we call the *non-collision* property stated in Property 1.

Property 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.*

Note that if we can guarantee the non-collision property, then we can use $\mathcal{F}_{\text{select-semi}}$ to generate the input sharing we want. 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).

Note that this requirement can be met without loss of generality 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. In the next subsection, we will discuss how to evaluate fan-out gates efficiently. 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 property we want.

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. We further assume that the number of the input wires and the number of the output wires of each layer are multiples of k, where recall that k is the number of secrets we can store in a single packed Shamir sharing. In Section 5, we will show how to compile a general circuit to meet this requirement.

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 4. *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 4 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.

Proof of Theorem 4. 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 1, there exists a perfect matching in G. Suppose $(1, \ell_1), (2, \ell_2), \ldots, (m, \ell_m)$ 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 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 $k \geq 1$. $\square$

4.4 Handling Fan-out Gates

In the last subsection, we discussed how to prepare the input sharings for multiplication gates and addition gates. Our solution requires 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 inserting fan-out gates in each layer, which copy each output wire the number of times it is used in later layers. Specifically, we consider a functionality $\mathcal{F}_{\text{fan-out-semi}}$ which takes as input a degree-t packed Shamir sharing of $\boldsymbol{x} = (x_1, x_2, \ldots, x_k) \in \mathbb{F}^k$ 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, and outputs $\frac{n_1+n_2+\ldots+n_k}{k}$ degree-t packed Shamir sharings which contain n_i copies of the value x_i for all $i \in \{1, 2, \ldots, k\}$. We assume that $\sum_{i=1}^{k} n_i$ is a multiple of k. In Section 5, we will show how to compile a general circuit to meet this requirement. The description of $\mathcal{F}_{\text{fan-out-semi}}$ appears in Functionality 15.

Let $[\boldsymbol{y}] \in \{[\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(m)}]\}$. We will focus on how to generate $[\boldsymbol{y}]$ from $[\boldsymbol{x}]$ in the following. $\mathcal{F}_{\text{fan-out-semi}}$ can be realized by generating $[\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(m)}]$ one by one.

Observe that $\boldsymbol{y}$ has the following form:

- $\boldsymbol{y}$ is a composition of ℓ vectors $\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(\ell)}$;
- each vector $\boldsymbol{y}^{(i)}$ only contains the same value;

Functionality 15: $\mathcal{F}_{\text{fan-out-semi}}$

1. $\mathcal{F}_{\text{fan-out-semi}}$ receives from honest parties the shares of $[\boldsymbol{x}]$ and a vector $(n_1, n_2, \ldots, n_k)$.

2. $\mathcal{F}_{\text{fan-out-semi}}$ reconstructs the secrets $\boldsymbol{x} = (x_1, x_2, \ldots, x_k)$. Then $\mathcal{F}_{\text{fan-out-semi}}$ initiates an empty list L. From $i = 1$ to k, $\mathcal{F}_{\text{fan-out-semi}}$ inserts n_i times of x_i into L.

3. Let $m = \frac{n_1+n_2+\ldots+n_k}{k}$. From $i = 1$ to m,

   (a) $\mathcal{F}_{\text{fan-out-semi}}$ sets $\boldsymbol{x}^{(i)}$ to be the vector of the first k elements in L, and then removes the first k elements in L.

   (b) $\mathcal{F}_{\text{fan-out-semi}}$ receives from the adversary a set of shares $\{s_j^{(i)}\}_{j\in\mathcal{C}}$. $\mathcal{F}_{\text{fan-out-semi}}$ generates a degree-t packed Shamir sharing $[\boldsymbol{x}^{(i)}]$ such that the j-th share of $[\boldsymbol{x}^{(i)}]$ is $s_j^{(i)}$.

   (c) $\mathcal{F}_{\text{fan-out-semi}}$ distributes the shares of $[\boldsymbol{x}^{(i)}]$ to honest parties.

- if let v_i be the value in $\boldsymbol{y}^{(i)}$, then $v_1, v_2, \ldots, v_\ell$ are ℓ distinct values.

This is because, in $\mathcal{F}_{\text{fan-out-semi}}$, we gather the same values when generating the list L. Therefore, for each vector $\boldsymbol{y} \in \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$, the entries that have the same value are also gathered.

For all $i \in \{1, 2, \ldots, \ell\}$, let w_i be the first position of $\boldsymbol{y}$ where v_i appears. Let $w_{\ell+1} = k + 1$. Then for all $i \in \{1, 2, \ldots, \ell\}$, entries with the indices between w_i and $w_{i+1} - 1$ are equal to v_i. To obtain $[\boldsymbol{y}]$ from $[\boldsymbol{x}]$, our idea is to do the following two steps:

1. All parties first permute the secrets of $[\boldsymbol{x}]$ such that for all $i \in \{1, 2, \ldots, \ell\}$, v_i is at position w_i. Let $[\boldsymbol{x}']$ denote the new sharing.

2. For all $i \in \{1, 2, \ldots, \ell\}$ and $w_i \leq j \leq w_{i+1} - 1$, all parties change the j-th secret of $\boldsymbol{x}'$ to be the same as the w_i-th secret of $\boldsymbol{x}'$. Note that the result will be a sharing of $\boldsymbol{y}$.

For Step 1, we can use $\mathcal{F}_{\text{permute-semi}}$ introduced in Section 4.2. For Step 2, relying on the techniques in [DIK10], it can be done by the following steps:

1. All parties prepare two random degree-t packed Shamir sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$, such that $\boldsymbol{r}$ is a uniform vector in $\mathbb{F}^k$, and for all $i \in \{1, 2, \ldots, \ell\}$ and $w_i \leq j \leq w_{i+1} - 1$, $\tilde{r}_j = r_{w_i}$.

2. All parties locally compute $[\boldsymbol{e}] := [\boldsymbol{x}'] + [\boldsymbol{r}]$.

3. All parties send their shares of $[\boldsymbol{e}]$ to the first party P_1.

4. P_1 reconstructs the secrets $\boldsymbol{e}$. Then P_1 sets $\tilde{\boldsymbol{e}}$ to be a vector in $\mathbb{F}^k$ such that for all $i \in \{1, 2, \ldots, \ell\}$ and $w_i \leq j \leq w_{i+1} - 1$, $\tilde{e}_j = e_{w_i}$. P_1 generates a random degree-t packed Shamir sharing $[\tilde{\boldsymbol{e}}]$ and distributes the shares to other parties.

5. All parties locally compute $[\boldsymbol{y}] := [\tilde{\boldsymbol{e}}] - [\tilde{\boldsymbol{r}}]$.

As for the correctness, note that $\boldsymbol{e} = \boldsymbol{x}' + \boldsymbol{r}$. Therefore, for all $i \in \{1, 2, \ldots, \ell\}$, $e_{w_i} = x'_{w_i} + r_{w_i} = v_i + r_{w_i}$. In Step 4, we have $\tilde{e}_j = e_{w_i} = v_i + r_{w_i}$ for all $i \in \{1, 2, \ldots, \ell\}$ and $w_i \leq j \leq w_{i+1} - 1$. In the last step, for all $i \in \{1, 2, \ldots, \ell\}$ and $w_i \leq j \leq w_{i+1} - 1$, $\tilde{e}_j - \tilde{r}_j = v_i + r_{w_i} - r_{w_i} = v_i = y_j$.

As in Section 4.2, the main issue of this approach is how to *efficiently* prepare a pair of random sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$. Indeed, such a pair of random sharings can be prepared using $\mathcal{F}_{\text{rand}}$ with a suitable $\mathbb{F}$-linear secret sharing scheme. However, for different structures of $\boldsymbol{y}$, the $\mathbb{F}$-linear secret sharing schemes used in $\mathcal{F}_{\text{rand}}$ are also different. Although the amortized communication complexity of the implementation of $\mathcal{F}_{\text{rand}}$ is $O(n)$

field elements, it has an overhead of $O(n^3 \cdot \kappa)$ field elements which is independent of the number of sharings we want to prepare. In the worst case where each time we need to prepare a different kind of random sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$, the overhead of obtaining $[\boldsymbol{y}]$ from $[\boldsymbol{x}]$ becomes $O(n^3 \cdot \kappa)$, which eliminates the benefit we gain when using the packed Shamir sharing scheme.

In [DIK10], they use a different way to handle the fan-out gate which increases the depth of the circuit by a factor of $O(\log |C|)$, where $|C|$ is the circuit size. We will use the same idea as that in Section 4.2 to prepare the random sharings we want.

Using $\mathcal{F}_{\text{select-semi}}$ to Generate Random Sharings for Copying Secrets. For all $i_1, i_2 \in \{1, 2, \dots, k\}$, we say a pair of degree-t packed Shamir sharings $([\boldsymbol{x}], [\boldsymbol{y}])$ contains an (i_1, i_2)-block if for all $i_1 \le j \le i_2$ the secrets of these two sharings satisfy that $y_j = x_{i_1}$. We say a point $j \in \{1, 2, \dots, k\}$ is covered by an (i_1, i_2)-block if $i_1 \le j \le i_2$. A pattern π is defined to be a list of blocks such that for all $j \in \{1, 2, \dots, k\}$, j is covered by exactly one block in π. Then, the random sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ we want to prepare correspond to a pattern π which contains the ℓ blocks: a $(w_1, w_2 - 1)$-block, a $(w_2, w_3 - 1)$-block, $\dots$, and a $(w_\ell, w_{\ell+1} - 1)$-block. Therefore, the problem we want to solve is to prepare a pair of random sharings for a pattern π.

Let m denote the number of patterns we want to prepare random sharings for. These patterns are denoted by $\pi_1, \pi_2, \dots, \pi_m$. Our idea is as follows:

1. We first find m patterns $\rho_1, \rho_2, \dots, \rho_m$ such that:
   - For all $i, j \in \{1, 2, \dots, k\}$, the number of patterns in $\{\pi_1, \pi_2, \dots, \pi_m\}$ that contain an (i, j)-block is equal to the number of patterns in $\{\rho_1, \rho_2, \dots, \rho_m\}$ that contain an (i, j)-block.
2. All parties prepare random sharings for patterns $\rho_1, \rho_2, \dots, \rho_m$.
3. From $i = 1$ to m, a pair of random sharings for the pattern π_i can be prepared as follows: for all (j_1, j_2)-block in π_i, all parties use $\mathcal{F}_{\text{select-semi}}$ to choose the first unused (j_1, j_2)-block from the random sharings for $\rho_1, \rho_2, \dots, \rho_m$.

The major benefit of this approach is that *we can limit the number of different patterns in* $\{\rho_1, \rho_2, \dots, \rho_m\}$ as we show below. More concretely, we will prove the following theorem:

Theorem 5. *Let $m, k \ge 1$ be integers. Let (i, j)-block and pattern be defined as above. For all m patterns $\pi_1, \pi_2, \dots, \pi_m$, there exists m patterns $\rho_1, \rho_2, \dots, \rho_m$ such that:*

- *For all $i, j \in \{1, 2, \dots, k\}$, the number of patterns $\pi \in \{\pi_1, \pi_2, \dots, \pi_m\}$ such that (i, j)-block is in π is the same as the number of patterns $\rho \in \{\rho_1, \rho_2, \dots, \rho_m\}$ such that (i, j)-block is in ρ.*
- *$\rho_1, \rho_2, \dots, \rho_m$ contain at most k^2 different patterns.*

Moreover, $\rho_1, \rho_2, \dots, \rho_m$ can be found within polynomial time given $\pi_1, \pi_2, \dots, \pi_m$.

Recall that the issue of using $\mathcal{F}_{\text{rand}}$ to prepare random sharings for $\pi_1, \pi_2, \dots, \pi_m$ is that there will be an overhead $O(n^3 \cdot \kappa)$ per different pattern in $\pi_1, \pi_2, \dots, \pi_m$. In the worst case, this overhead can be as large as $O(n^3 \cdot \kappa \cdot m)$, which eliminates the benefit of using the packed Shamir sharing scheme. Relying on $\mathcal{F}_{\text{select-semi}}$, we only need to prepare random sharings for patterns $\rho_1, \dots, \rho_m$, which contain at most $k^2 \le n^2$ different patterns. In this way, the overhead is upper-bounded by $O(n^5 \cdot \kappa)$, which is independent of the number of patterns we want to prepare random sharings for.

Proof of Theorem 5. Recall that we say a value $j \in \{1, 2, \dots, k\}$ is covered by an (i_1, i_2)-block if $i_1 \le j \le i_2$, and a pattern is defined to be a set of blocks such that each value in $\{1, 2, \dots, k\}$ is covered by exactly one block. One may view each block as a small segment, and a pattern as a segment coverage of the segment $[1, k]$.

Let L be a two-dimensional list with indices $i, j \in \{1, 2, \dots, k\}$, such that $L(i, j)$ is the number of patterns $\pi \in \{\pi_1, \pi_2, \dots, \pi_m\}$ such that (i, j)-block is in π. Then, L is a list of demand. We say a two-dimensional list L a *balanced list* if

- for all $j \in \{1, 2, \ldots, k\}$, the summation $\sum_{i_1 \le j \le i_2} L(i_1, i_2)$ is the same.

The summation $\sum_{i_1 \le j \le i_2} L(i_1, i_2)$ represents the number of blocks that cover j. Therefore, a balanced list corresponds to a set of blocks such that for all $j \in \{1, 2, \ldots, k\}$, j is covered by the same number of times.

We first prove the following property of a balanced list L:

Property 2 (Pattern)**.** *For all $i \le j < k$, if $L(i, j) \ge 1$, then there exists $j < j' \le k$ such that $L(j+1, j') \ge 1$.*

In another word, this property says that for the set of blocks that L corresponds to, if there exists an (i, j)-block where $j < k$, then there exists another block which starts from $j + 1$. To see this, note that L is a balanced list. Therefore, j and $j + 1$ are covered by the same number of blocks. Since we already have a block which covers j but not $j + 1$, there must exist a block which covers $j + 1$ but not j, which means that this block starts from $j + 1$.

Consider the following process to find the patterns $\rho_1, \rho_2, \ldots, \rho_m$.

1. Initially set the list of patterns Q to be empty. Let $L' := L$. Note that L' is a balanced list.
2. While L' contains a non-zero entry, run the following steps:
   (a) Initially set ρ to be an empty set of blocks. Set $i = 1$ which represents the first value which is not covered by the blocks in ρ.
   (b) While $i \le k$:
      i. Find the first entry $j \ge i$ such that $L'(i, j) \ge 1$.
      ii. Insert (i, j)-block in ρ.
      iii. Set $i := j + 1$.
   (c) Let $n' = \min\{L'(i_1, i_2) |\ (i_1, i_2)\text{-block is in } \rho\}$. For all $i_1, i_2 \in \{1, 2, \ldots, k\}$ such that (i_1, i_2)-block is in ρ, set $L'(i_1, i_2) := L'(i_1, i_2) - n'$.
   (d) Insert n' times of ρ into the list Q.
3. Output the patterns in Q.

We first show that the above process will finally terminate. In the beginning, $L' = L$ is a balanced list. We maintain this property in each iteration of Step 2. At the beginning of Step 2, since L' is a non-zero balanced list, the value 1 is covered by at least 1 block. Therefore, we can find $j \ge 1$ such that $L'(1, j) \ge 1$. For $1 < i \le k$, since we have found a block which ends at the value $i - 1$, by the property of a balanced list, there exists $j \ge i$ such that $L'(i, j) \ge 1$. This process ends when $i = k + 1$. Note that the blocks we select form a pattern. In Step 2.(c), n' is the largest number of the patterns ρ we can have. Having more than n' times of pattern ρ will lead to at least one block having more supply than demand. Since ρ is a pattern, each value is covered by exactly one block in ρ. Therefore, after updating the list of demand L', L' is still a balanced list. Note that the summation of all the entries in L' decreases in each iteration. We conclude that the above process will finally terminate.

Now we show that the number of different patterns in Q is bounded by k^2. It is sufficient to show that Step 2 will terminate within k^2 rounds since in each round we only insert the same type of patterns into Q. To this end, we count the number of 0-entries in L'. Note that in each round, the way we choose n' in Step 2.(c) guarantees that at least one entry of L' will become 0 after updating L' in Step 2.(c). Therefore, the number of 0-entries in L' increases at least by 1 in each round. Since there are k^2 entries in L', Step 2 will terminate within k^2 rounds. □

Preparing Random Sharings for Different Patterns. We are ready to introduce the functionality and its implementation for preparing random sharings for different patterns. The functionality $\mathcal{F}_{\text{rand-pattern-semi}}$ appears in Functionality 16.

For a fixed pattern π, we show how to use $\mathcal{F}_{\text{rand}}$ to prepare a pair of random sharings for π in Appendix B.4. The communication complexity of preparing m pairs of random sharings for a fixed pattern π is

Functionality 16: $\mathcal{F}_{\text{rand-pattern-semi}}$

1. $\mathcal{F}_{\text{rand-pattern-semi}}$ receives from honest parties m patterns $\pi_1, \pi_2, \ldots, \pi_m$.

2. For all $i \in [m]$, $\mathcal{F}_{\text{rand-pattern-semi}}$ receives from the adversary a set of shares $\{(u_j^{(i)}, v_j^{(i)})\}_{j\in\mathcal{C}}$. $\mathcal{F}_{\text{rand-pattern-semi}}$ samples a random vector $\boldsymbol{r}^{(i)} \in \mathbb{F}^k$ and computes a vector $\tilde{\boldsymbol{r}}^{(i)}$ such that for all (i_1, i_2)-block in π_i and $i_1 \leq j \leq i_2$, $\tilde{r}_j^{(i)} = r_{i_1}^{(i)}$. $\mathcal{F}_{\text{rand-pattern-semi}}$ samples two degree-t packed Shamir sharings $([\boldsymbol{r}^{(i)}], [\tilde{\boldsymbol{r}}^{(i)}])$ such that for all $P_j \in \mathcal{C}$, the j-th share of $([\boldsymbol{r}^{(i)}], [\tilde{\boldsymbol{r}}^{(i)}])$ is $(u_j^{(i)}, v_j^{(i)})$.

3. For all $i \in [m]$, $\mathcal{F}_{\text{rand-pattern-semi}}$ distributes the shares of $([\boldsymbol{r}^{(i)}], [\tilde{\boldsymbol{r}}^{(i)}])$ to honest parties.

$O(m \cdot n + n^3 \cdot \kappa)$ elements in $\mathbb{F}$. We describe the protocol for $\mathcal{F}_{\text{rand-pattern-semi}}$ in Protocol 17. The communication complexity of using Rand-Pattern to prepare random sharings for m patterns is $O(m \cdot n + n^5 \cdot \kappa)$ field elements.

Lemma 7. *Protocol* Rand-Pattern *securely computes* $\mathcal{F}_{\textit{rand-pattern-semi}}$ *in the* $(\mathcal{F}_{\textit{rand}}, \mathcal{F}_{\textit{select-semi}})$*-hybrid model against a semi-honest adversary who controls* $t' = t - k + 1$ *parties.*

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

In Step 1 and Step 2, $\mathcal{S}$ honestly follows the protocol and computes the patterns $\rho_1, \rho_2, \ldots, \rho_m$. In Step 3, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ when generating the random sharings $([\boldsymbol{r}^{(i)}], [\tilde{\boldsymbol{r}}^{(i)}])$ for each pattern ρ_i. Since the number of corrupted parties is $t - k + 1$, by the property of the degree-t packed Shamir sharing scheme, the secrets $\boldsymbol{r}^{(i)}$ are uniform and independent of the shares held by corrupted parties. In Step 4, $\mathcal{S}$ honestly constructs the lists. In Step 5, $\mathcal{S}$ emulates $\mathcal{F}_{\text{select-semi}}$ and receives the shares of corrupted parties for all $([\boldsymbol{v}^{(\ell)}], [\tilde{\boldsymbol{v}}^{(\ell)}])$. Then, $\mathcal{S}$ sends these shares to $\mathcal{F}_{\text{rand-pattern-semi}}$.

Note that through the protocol, corrupted parties do not receive any messages from honest parties. Therefore, it is sufficient to argue that the distribution of the output in both worlds are identical. First, for all $\ell \in \{1, 2, \ldots, m\}$, the shares of $[\boldsymbol{v}^{(\ell)}]$ and $[\tilde{\boldsymbol{v}}^{(\ell)}]$ held by corrupted parties are chosen by themselves in both worlds. For the secrets $\boldsymbol{v}^{(\ell)}$ and $\tilde{\boldsymbol{v}}^{(\ell)}$, let s denote the number of blocks in π_ℓ, and the blocks in π_ℓ are denoted by $(w_1, w_2 - 1)$-block, $(w_2, w_3 - 1)$-block, $\ldots$, $(w_s, w_{s+1} - 1)$-block, where $w_1 = 1$, $w_{s+1} = k + 1$ and $w_1 < w_2 < \ldots < w_{s+1}$. Then in the ideal world, $\boldsymbol{v}^{(\ell)}$ is a uniform vector in $\mathbb{F}^k$ and $\tilde{\boldsymbol{v}}^{(\ell)}$ satisfies that for all $i \in \{1, 2, \ldots, s\}$ and $w_i \leq j < w_{i+1}$, $\tilde{v}_j^{(\ell)} = v_{w_i}^{(\ell)}$. In the real world, there is a one-to-one map from $\{\boldsymbol{r}^{(\ell)}\}_{\ell\in[m]}$ to $\{\boldsymbol{v}^{(\ell)}\}_{\ell\in[m]}$. Therefore each vector $\boldsymbol{v}^{(\ell)}$ is uniform. As for $\tilde{\boldsymbol{v}}^{(\ell)}$, note that in Step 5, Case 2.2, for all $i \in \{1, 2, \ldots, s\}$ and $w_i \leq j < w_{i+1}$, $v_j^{(\ell)} = r_j^{(\ell_i)}$ and $\tilde{v}_j^{(\ell)} = \tilde{r}_j^{(\ell_i)} = r_{w_i}^{(\ell_i)} = v_{w_i}^{(\ell)}$, where the second equation is because $([\boldsymbol{r}^{(\ell_i)}], [\tilde{\boldsymbol{r}}^{(\ell_i)}])$ contains a $(w_i, w_{i+1} - 1)$-block. Therefore, the distribution of the output in both worlds are identical. $\square$

Realizing $\mathcal{F}_{\textbf{fan-out-semi}}$. Now we are ready to present the protocol for $\mathcal{F}_{\text{fan-out-semi}}$. Recall that our idea for $\mathcal{F}_{\text{fan-out-semi}}$ is to generate the output sharings one by one. For each output sharing, we do the following two steps:

1. In the first step, we permute the input sharing such that each secret we need to copy for this output sharing is in the first position of each block.

2. In the second step, we use $\mathcal{F}_{\text{rand-pattern-semi}}$ to prepare the random sharings we want and then follow the techniques in [DIK10] as discussed above.

Similarly to Permute, we will prepare a random degree-$2t$ packed Shamir sharing of $\mathbf{0} \in \mathbb{F}^k$, which is used as a random mask for the shares of honest parties (see the proof of Lemma 8). This is not needed

Protocol 17: Rand-Pattern

1. Let $\pi_1, \pi_2, \ldots, \pi_m$ be the patterns that all parties want to prepare random sharings for.
2. All parties use a deterministic algorithm that all parties agree on to compute m patterns $\rho_1, \rho_2, \ldots, \rho_m$ such that
   - For all $i, j \in \{1, 2, \ldots, k\}$, the number of patterns $\pi \in \{\pi_1, \pi_2, \ldots, \pi_m\}$ such that (i, j)-block is in π is the same as the number of patterns $\rho \in \{\rho_1, \rho_2, \ldots, \rho_m\}$ such that (i, j)-block is in ρ.
   - $\rho_1, \rho_2, \ldots, \rho_m$ contain at most k^2 different patterns.

   The existence of such an algorithm is guaranteed by Theorem 5.
3. Suppose $\rho'_1, \rho'_2, \ldots, \rho'_{k^2}$ denote the different patterns in $\rho_1, \rho_2, \ldots, \rho_m$. For all $i \in \{1, 2, \ldots, k^2\}$, let n'_i denote the number of times that ρ'_i appears in $\rho_1, \rho_2, \ldots, \rho_m$. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare n'_i pairs of random sharings for the pattern ρ'_i for all $i \in \{1, 2, \ldots, k^2\}$. Note that we have prepared a pair of random sharings for pattern ρ_i for all $i \in [m]$. Let $([\boldsymbol{r}^{(i)}], [\tilde{\boldsymbol{r}}^{(i)}])$ denote the random sharings for the pattern ρ_i.
4. For all $i, j \in \{1, 2, \ldots, k\}$, all parties initiate an empty list $L_{i,j}$. From $\ell = 1$ to m, for all $i, j \in \{1, 2, \ldots, k\}$, if $([\boldsymbol{r}^{(\ell)}], [\tilde{\boldsymbol{r}}^{(\ell)}])$ contains an (i, j)-block, all parties insert $([\boldsymbol{r}^{(\ell)}], [\tilde{\boldsymbol{r}}^{(\ell)}])$ into the list $L_{i,j}$.
5. From $\ell = 1$ to m, all parties prepare a pair of random sharings for π_ℓ as follows:
   - Let s denote the number of blocks in π_ℓ. The blocks in π_ℓ are denoted by $(w_1, w_2 - 1)$-block, $(w_2, w_3 - 1)$-block, $\ldots$, $(w_s, w_{s+1} - 1)$-block, where $w_1 = 1$, $w_{s+1} = k + 1$ and $w_1 < w_2 < \ldots < w_{s+1}$. From $i = 1$ to s, let $([\boldsymbol{r}^{(\ell_i)}], [\tilde{\boldsymbol{r}}^{(\ell_i)}])$ denote the first pair of sharings in the list $L_{w_i, w_{i+1}-1}$, and then remove it from $L_{w_i, w_{i+1}-1}$. Note that $([\boldsymbol{r}^{(\ell_i)}], [\tilde{\boldsymbol{r}}^{(\ell_i)}])$ contains a $(w_i, w_{i+1} - 1)$-block, which is not used when preparing random sharings for $\pi_1, \pi_2, \ldots, \pi_{\ell-1}$.
   - Let I denote the identity permutation over $\{1, 2, \ldots, k\}$.
     - All parties initiate an empty list Q. For all $j \in \{1, 2, \ldots, k\}$, let i_j denote the index such that $w_{i_j} \le j < w_{i_j+1}$. From $j = 1$ to k, all parties insert $[\boldsymbol{r}^{(\ell_{i_j})}]$ into Q. Then, all parties invoke $\mathcal{F}_{\text{select-semi}}$ with the degree-t packed Shamir sharings in Q and the permutation I. The output is denoted by $[\boldsymbol{v}^{(\ell)}]$.
     - All parties initiate an empty list Q'. From $j = 1$ to k, all parties insert $[\tilde{\boldsymbol{r}}^{(\ell_{i_j})}]$ into Q'. Then, all parties invoke $\mathcal{F}_{\text{select-semi}}$ with the degree-t packed Shamir sharings in Q' and the permutation I. The output is denoted by $[\tilde{\boldsymbol{v}}^{(\ell)}]$. Note that for all $i \in \{1, 2, \ldots, s\}$ and $w_i \le j < w_{i+1}$, $v_j^{(\ell)} = r_j^{(\ell_i)}$ and $\tilde{v}_j^{(\ell)} = \tilde{r}_j^{(\ell_i)} = r_{w_i}^{(\ell_i)} = v_{w_i}^{(\ell)}$.
6. All parties take $([\boldsymbol{v}^{(1)}], [\tilde{\boldsymbol{v}}^{(1)}]), ([\boldsymbol{v}^{(2)}], [\tilde{\boldsymbol{v}}^{(2)}]), \ldots, ([\boldsymbol{v}^{(\ell)}], [\tilde{\boldsymbol{v}}^{(\ell)}])$ as output.

for semi-honest security but will be helpful when we consider a fully malicious adversary at a later point. The description of Fan-out appears in Protocol 18. As for the communication complexity of Fan-out, we measure it by the number of output sharings. This is because, in Fan-out, the first two steps do not require any communication and are used to determine the structures of the secrets of the output sharings. In Step 3, output sharings are prepared one by one. Therefore, the communication complexity only depends on the number of output sharings *even if the output sharings are coming from different invocations of* Fan-out *with* *different input sharings.* The communication complexity of using Fan-out to generate m output sharings is $O(m \cdot n + n^5 \cdot \kappa)$ field elements.

Protocol 18: Fan-out

1. Let $[\boldsymbol{x}]$ denote the input degree-t packed Shamir sharing and $(n_1, n_2, \ldots, n_k)$ denote the vector in $\mathbb{N}^k$ where n_i is the number of times of the i-th entry of $\boldsymbol{x}$ that all parties want to copy.

2. All parties run the following steps to determine the secrets of each output sharing:

   (a) Initially, L is set to be an empty list. From $i = 1$ to k, insert n_i times of x_i into L.

   (b) Let $m = \frac{n_1+n_2+\ldots+n_k}{k}$. From $i = 1$ to m, let $\boldsymbol{x}^{(i)}$ be the vector of the first k elements in L, and then removes the first k elements in L. Then, $\boldsymbol{x}^{(i)}$ are the secrets of the i-th output degree-t packed Shamir sharing.

3. For each of $\boldsymbol{y} \in \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$, all parties run the following steps to generate a degree-t packed Shamir sharing of $\boldsymbol{y}$.

   (a) Let s denote the number of different values in $\boldsymbol{y}$. These different values are denoted by $v_1, v_2, \ldots, v_s$. For all $i \in \{1, 2, \ldots, s\}$, let w_i denote the index of the first value of $\boldsymbol{y}$ that is equal to v_i. Let $p(\cdot)$ be a permutation over $\{1, 2, \ldots, k\}$ that all parties agree on such that, after computing $\boldsymbol{x}' = (x_{p(1)}, x_{p(2)}, \ldots, x_{p(k)})$, $x'_{w_i} = v_i$ for all $i \in \{1, 2, \ldots, s\}$.

   (b) All parties invoke $\mathcal{F}_{\text{permute-semi}}$ with input sharing $[\boldsymbol{x}]$ and the permutation p. Let $[\boldsymbol{x}']$ denote the output.

   (c) Let π be a pattern which contains $(w_1, w_2 - 1)$-block, $(w_2, w_3 - 1)$-block, $\ldots$, $(w_s, w_{s+1} - 1)$-block, where $w_{s+1} = k + 1$. All parties invoke $\mathcal{F}_{\text{rand-pattern-semi}}$ with π to prepare a pair of random sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ such that $\boldsymbol{r}$ is uniform in $\mathbb{F}^k$ and for all $(w_i, w_{i+1} - 1)$-block in π and $w_i \le j < w_{i+1}$, $\tilde{r}_j = r_{w_i}$.

   (d) All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare a random degree-$2t$ packed Shamir sharing $\langle \mathbf{0} \rangle$.

   (e) All parties locally compute $\langle \boldsymbol{e} \rangle := [\boldsymbol{x}'] + [\boldsymbol{r}] + \langle \mathbf{0} \rangle$.

   (f) All parties send their shares of $\langle \boldsymbol{e} \rangle$ to the first party P_1.

   (g) P_1 reconstructs the secrets $\boldsymbol{e}$, and computes $\tilde{\boldsymbol{e}}$ such that for all $(w_i, w_{i+1} - 1)$-block in π and $w_i \le j < w_{i+1}$, $\tilde{e}_j = e_{w_i}$. Then P_1 generates a random degree-t packed Shamir sharing $[\tilde{\boldsymbol{e}}]$, and distributes the shares to other parties.

   (h) All parties locally compute $[\boldsymbol{y}] := [\tilde{\boldsymbol{e}}] - [\tilde{\boldsymbol{r}}]$.

Lemma 8. *Protocol* Fan-out *securely computes $\mathcal{F}_{\textit{fan-out-semi}}$ in the $(\mathcal{F}_{\textit{rand}}, \mathcal{F}_{\textit{rand-pattern-semi}})$-hybrid model against a semi-honest adversary who controls $t' = t - k + 1$ parties.*

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

Recall that in Lemma 6, we have shown that, for a random degree-t packed Shamir sharing $[\boldsymbol{r}]$ and a random degree-$2t$ packed Shamir sharing of $\mathbf{0} \in \mathbb{F}^k$, given the shares of $[\boldsymbol{r}]$ and $\langle \mathbf{0} \rangle$ held by corrupted parties, the shares of $\langle \boldsymbol{r} \rangle := [\boldsymbol{r}] + \langle \mathbf{0} \rangle$ held by honest parties are uniform.

We first prove the correctness of Fan-out. In Step 1 and Step 2, all parties determine the secrets of each output degree-t packed Shamir sharing. The procedure is identical to that in $\mathcal{F}_{\text{fan-out-semi}}$. Therefore, it is sufficient to show that in Step 3, all parties can obtain each output sharing correctly.

In Step 3, for each $\boldsymbol{y} \in \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$, $\boldsymbol{y}$ is in the form that (1) $\boldsymbol{y}$ is a composition of ℓ small vectors $\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, \ldots, \boldsymbol{y}^{(\ell)}$; (2) each small vector $\boldsymbol{y}^{(i)}$ only contains the same value; and (3) if let v_i be the value in $\boldsymbol{y}^{(i)}$, then $v_1, v_2, \ldots, v_\ell$ are ℓ distinct values. This is because we gather the same values when we construct the list L in Step 2. Therefore, for each $\boldsymbol{y} \in \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$, the entries that have the same value are also gathered. In Step 3.(b), we permute the secrets of $[\boldsymbol{x}]$ such that the secrets $\boldsymbol{x}'$ of the output sharing satisfies that $x'_{w_i} = v_i = y_{w_i}$ for all $i \in \{1, 2, \ldots, s\}$. Note that for all $i \in \{1, 2, \ldots, s\}$ and $w_i \leq j < w_{i+1}$, $y_j = y_{w_i} = v_i$. To see that all parties obtain the correct secrets $\boldsymbol{y}$ when running Step 3.(e) to Step 3.(h), note that $\boldsymbol{e} = \boldsymbol{x}' + \boldsymbol{r}$. Therefore $e_{w_i} = x'_{w_i} + r_{w_i} = v_i + r_{w_i}$ for all $i \in \{1, 2, \ldots, s\}$. Since for all $i \in \{1, 2, \ldots, s\}$ and $w_i \leq j < w_{i+1}$, $\tilde{e}_j = e_{w_i}$ and $\tilde{r}_j = r_{w_i}$, we have $y_j = \tilde{e}_j - \tilde{r}_j = e_{w_i} - r_{w_i} = v_i$.

Now we describe the construction of $\mathcal{S}$. $\mathcal{S}$ determines the secrets of each output degree-t packed Shamir sharing by honestly following the protocol. In Step 3.(b), $\mathcal{S}$ emulates $\mathcal{F}_{\text{permute-semi}}$. In Step 3.(c) and Step 3.(d), $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand-pattern-semi}}$ and $\mathcal{F}_{\text{rand}}$, and receives the shares of $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ and $\langle \boldsymbol{0} \rangle$ held by corrupted parties. In Step 3.(e), for each honest party, $\mathcal{S}$ generates a random element in $\mathbb{F}$ as its share of $\langle \boldsymbol{e} \rangle$. As we argued above, the share of $\langle \boldsymbol{r} \rangle := [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle$ held by each honest party is uniform. Therefore, the share of $\langle \boldsymbol{e} \rangle$ held by each honest party is also uniform. In Step 3.(f), Step 3.(g) and Step 3.(h), $\mathcal{S}$ honestly follows the protocol. At the end of the protocol, $\mathcal{S}$ computes the shares of $[\boldsymbol{y}]$ held by corrupted parties. This can be computed by using the shares of $[\tilde{\boldsymbol{e}}]$ and $[\tilde{\boldsymbol{r}}]$ held by corrupted parties, where

- the shares of $[\tilde{\boldsymbol{e}}]$ held by corrupted parties can be computed by the shares of honest parties $\mathcal{S}$ learnt in Step 3.(g),
- and the shares of $[\tilde{\boldsymbol{r}}]$ held by corrupted parties are received when $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand-pattern-semi}}$ in Step 3.(c).

Then $\mathcal{S}$ sends the shares of $[\boldsymbol{y}]$ held by corrupted parties to $\mathcal{F}_{\text{fan-out-semi}}$.

It is clear that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. As for the output, note that the whole sharing $[\boldsymbol{y}]$ is determined by the secrets $\boldsymbol{y}$ and the shares of corrupted parties. Since $\boldsymbol{y}$ is determined in the same way in both worlds, and the shares of $[\boldsymbol{y}]$ of corrupted parties are chosen by the adversary in both worlds, the output of honest parties is identical in both worlds. $\square$

Remark 3. *We note that the functionality $\mathcal{F}_{\text{rand-pattern-semi}}$ needs to know all the patterns before generating the random sharings. In the real execution, all the invocations of* FAN-OUT *will first run Step 3.(b) to submit the pattern to $\mathcal{F}_{\text{rand-pattern-semi}}$. This can be done since the patterns we need to prepare random sharings for are determined by the circuit and are independent of the inputs.*

5 Main Protocol - Against a Semi-honest Adversary

In this section, we will introduce our main protocol of using packed Shamir sharing to evaluate a general circuit C against a *semi-honest* adversary. We first discuss how to compile a general circuit to meet the requirements we assume in Section 4. Then we give the main protocol and analyze its security and communication complexity.

Recall that we are in the client-server model where there are c clients and $n = 2t + 1$ parties (servers). Recall that $1 \leq k \leq t$ is an integer. An adversary is allowed to corrupt $t' = t - k + 1$ parties. We will use the degree-t packed Shamir sharing scheme, which can store k secrets within one sharing. Recall that $\mathcal{C}$ denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

5.1 Transforming a General Circuit C

In Section 4, we assume that the circuit has the following properties:

- In the input layer and the output layer, the number of input gates belonging to each client and the number of output gates belonging to each client are multiples of k. In each intermediate layer, the number of addition gates and the number of multiplication gates are multiples of k. (See Section 4.1.)

- For the input layer and all intermediate layers, the number of output wires of each layer is a multiple of k. For the output layer and all intermediate layers, the number of input wires of each layer is a multiple of k. Moreover, each output wire is only used once as an input wire in a later layer so that there is a bijective map between the output wires and the input wires. (See Section 4.3.)
- During the computation, gates that have the same type (i.e., input gates belonging to the same client, output gates belonging to the same client, multiplication gates, addition gates) in each layer are divided into groups of k. Each group of gates are evaluated simultaneously. For the output wires of each group of gates, the number of times that those wires are used as input wires in later layers is a multiple of k. (See Section 4.4.)

Note that the second property is implied by the first property and the third property:

- As we argued in 4.3, without loss of generality, we assume that there is a fan-out gate right after each of input gates, multiplication gates, and addition gates, which copies the output wire the number of times that this output wire is used in later layers. In this way, every output wire is only used once as an input wire in a later layer.
- According to the first property, the number of addition gates and the number of multiplication gates in each intermediate layer are multiples of k. Therefore, the number of input wires of each intermediate layer is a multiple of k. Similarly, for the output layer, since the number of output gates belonging to each client is a multiple of k, the total number of input wires of the output layer is also a multiple of k.
- According to the third property, after grouping the gates that have the same type in each layer, the number of times that the output wires of each group of gates are used in later layers are multiples of k. Therefore, after using fan-out gates to copy each output wire the number of times that this wire is used in later layers, the number of output wires of each of the input layer and intermediate layers is a multiple of k.

Thus, it is sufficient to show the following theorem:

Theorem 6. *Given an arithmetic circuit C with input coming from c clients, there exists an efficient algorithm which takes C as input and outputs an arithmetic circuit C' with the following properties:*

- *For all input x, $C(x) = C'(x)$.*
- *In the input layer and the output layer, the number of input gates belonging to each client and the number of output gates belonging to each client are multiples of k. In each intermediate layer, the number of addition gates and the number of multiplication gates are multiples of k.*
- *After grouping the gates that have the same type in each layer, the number of times that the output wires of each group are used in later layers is a multiple of k.*
- *Circuit size: $|C'| = O(|C| + k \cdot (c + \mathsf{Depth}))$, where c is the number of clients that provide inputs and Depth is the depth of C.*

Proof. Let $\mathsf{Client}_1, \mathsf{Client}_2, \ldots, \mathsf{Client}_c$ denote the clients that provide inputs. We start by creating a virtual client Client_0. We will insert two new types of gates: input gates belonging to Client_0 and output gates belonging to Client_0. The input gates belonging to Client_0 are used to provide constant values for the computation. The output gates belonging to Client_0 are used to collect the wires that *will not be output to any clients.* Initially, for each input wire carrying a constant value in C, we insert an input gate belonging to Client_0 for this wire. Without loss of generality, we assume that each input of Client_0 is only used once in the circuit. To see this, for an input of Client_0 (which is known to all parties), if this input is used more than once, we can simply insert multiple input gates that take the same value. (Note that the same argument does not work for other clients since it allows a client to use different values for its input wires in C' which should have the same value in C.)

Step 1. For each i, let M_i and A_i denote the number of multiplication gates and the number of addition gates in layer i. We insert $\lceil \frac{M_i}{k} \rceil \cdot k - M_i$ multiplication gates and $\lceil \frac{A_i}{k} \rceil \cdot k - A_i$ addition gates in layer i. Each of these new gates takes two inputs from Client_0 and outputs to Client_0 as well. The inputs are set to be 0. Therefore, we insert 2 input gates belonging to Client_0 and one output gate belonging to Client_0 for each of the new multiplication gates and addition gates. In this way, the number of multiplication gates and the number of addition gates are multiples of k. Note that this step increases the circuit size by $O(k \cdot \mathsf{Depth})$.

Step 2. For each intermediate layer, we divide the multiplication gates and addition gates into groups of k respectively. For each group of gates, let $(w_1, w_2, \ldots, w_k)$ denote the output wires, and $(n_1, n_2, \ldots, n_k)$ denote the number of times that each wire is used in later layers. Let $n'_k = \lceil \frac{n_1+n_2+\ldots+n_k}{k} \rceil \cdot k - (n_1+n_2+\ldots+n_k)$. We insert n'_k output gates belonging to Client_0 which take w_k as the input wire. In this way, w_k is used $n_k + n'_k$ times, and the total number of times that $w_1, w_2, \ldots, w_k$ are used in later layers is a multiple of k. Note that each wire w_j is used at least once. Therefore, $n'_k < k \le n_1 + n_2 + \ldots + n_k$. It means that for the output wires of each group, we increase the number of times that these wires are used in later layers by at most a factor of 2. Therefore this step increases the circuit size by at most a factor of 2. It is convenient to think that for each group of gates, there is a fan-out gate which takes as input the output wires $(w_1, w_2, \ldots, w_k)$ and $(n_1, n_2, \ldots, n_k + n'_k)$, and outputs n_1 copies of w_1, n_2 copies of w_2, $\ldots$, and $n_k + n'_k$ copies of w_k. We consider the fan-out gates as a part of each intermediate layer.

Step 3. For each Client_i ($i \ge 1$), let I_i denote the number of input gates belonging to Client_i. We insert $\lceil \frac{I_i}{k} \rceil \cdot k - I_i$ input gates belonging to Client_i, which take 0 as input. We also insert the same number of output gates belonging to Client_0 which take these inputs as output. In this way, the number of input gates belonging to each Client_i is a multiple of k. Similarly, let O_i denote the number of output gates belonging to Client_i. We insert $\lceil \frac{O_i}{k} \rceil \cdot k - O_i$ input gates belonging to Client_0, which take 0 as input. We also insert the same number of output gates belonging to Client_i which take these inputs as output. In this way, the number of output gates belonging to each Client_i is a multiple of k. Note that this step increases the circuit size by $O(k \cdot c)$.

Step 4. For Client_0, let I_0 denote the number of input gates belonging to Client_0. We insert $\lceil \frac{I_0}{k} \rceil \cdot k - I_0$ input gates belonging to Client_0, which take 0 as input. We also create the same number of output gates belonging to Client_0 which take these inputs as output. In this way, the number of input gates belonging to Client_0 is a multiple of k. This step increases the circuit size by $O(k)$.

Step 5. For each Client_i, we divide the input gates belonging to Client_i into groups of k. Similarly to **Step 2**, for each group of gates, let $(w_1, w_2, \ldots, w_k)$ denote the output wires, and $(n_1, n_2, \ldots, n_k)$ denote the number of times that each wire is used in later layers. Let $n'_k = \lceil \frac{n_1+n_2+\ldots+n_k}{k} \rceil \cdot k - (n_1 + n_2 + \ldots + n_k)$. We insert n'_k output gates belonging to Client_0 which take w_k as the input wire. In this way, w_k is used $n_k + n'_k$ times, and the total number of times that $w_1, w_2, \ldots, w_k$ are used in later layers is a multiple of k. Together with **Step 2**, the circuit size is increased by at most a factor of 2. It is convenient to think that for each group of gates, there is a fan-out gate which takes as input the output wires $(w_1, w_2, \ldots, w_k)$ and $(n_1, n_2, \ldots, n_k + n'_k)$, and outputs n_1 copies of w_1, n_2 copies of w_2, $\ldots$, and $n_k + n'_k$ copies of w_k. We consider the fan-out gates as a part of the input layer. In this way, each output wire is only used once in later layers.

This completes the transformation of the circuit.

Now we show that the circuit after the transformation has the desired properties:

- For the first property, note that we do not change the original gates and wires in C. Therefore, it is sufficient to show that the gates and wires we add in the transformation do not change the functionality. For the input layer, we create several new input gates belonging to each Client_i ($i \ge 1$). These new gates directly connect to the output gates of Client_0, which means that these values are never used in the computation. For the output layer, we create several new output gates belonging to each Client_i ($i \ge 1$). These new gates directly take the value 0 from the input gates of Client_0, which do not affect the final result. Therefore, the first property holds.

- For the second and the third property, we verify them layer by layer:

  - For the input layer, by the way we insert new gates in **Step 3** and **Step 4**, the number of input gates belonging to each client is a multiple of k. By **Step 5**, the number of times that the output wires of each group of gates are used in later layers is a multiple of k.
  - For all intermediate layers, by the way we insert new gates in **Step 1**, the number of multiplication gates and the number of addition gates in each layer are multiples of k. By **Step 2**, the number of times that the output wires of each group of gates are used in later layers is a multiple of k.
  - For the output layer, we only need to verify that the number of output gates belonging to each client is a multiple of k. Note that it holds for $\mathsf{Client}_1, \mathsf{Client}_2, \ldots, \mathsf{Client}_c$ by the way we insert new gates in **Step 3**. As for the output gates belonging to Client_0, note that the total number of output wires of the input layer and all intermediate layers is a multiple of k, and the number of input wires of each intermediate layer is also a multiple of k. Therefore, the input wires of the output layer is a multiple of k, which means that the number of output gates is a multiple of k. Since the number of output gates belonging to each of $\mathsf{Client}_1, \mathsf{Client}_2, \ldots, \mathsf{Client}_c$ is a multiple of k. We conclude that the number of output gates belonging to Client_0 is also a multiple of k.

- In **Step 1**, the circuit size increases by $O(k \cdot \mathsf{Depth})$. In **Step 2** and **Step 5**, the circuit size increases by at most a factor of 2. In **Step 3**, the circuit size increases by $O(k \cdot c)$. In **Step 4**, the circuit size increases by $O(k)$. Thus, the size of the circuit after transformation is bounded by $O(|C| + k \cdot (c + \mathsf{Depth}))$.

□

5.2 Preprocessing Phase

In this part, we describe how parties preprocess the circuit before doing the computation. During the computation phase, a batch of k wire values are stored in a single packed Shamir sharing. The main task of the preprocessing phase is to determine how the wire values should be packed. Also, all parties need to compute a permutation for each output sharing using the algorithm in Theorem 4. These permutations are used to achieve the non-collision property. See Section 4.3 for more details. The preprocessing phase only depends on the circuit C and does not need any communication. The description of PREPROCESS appears in Protocol 19.

5.3 Main Protocol - Against Semi-honest Adversary

We are ready to introduce our main protocol. At a high-level, given the preprocessed circuit,

- all parties use $\mathcal{F}_{\text{input-semi}}, \mathcal{F}_{\text{output-semi}}, \mathcal{F}_{\text{mult-semi}}$ (see Section 4.1) to evaluate input gates, output gates, multiplication gates, and addition gates in each layer;
- for the input layer and all intermediate layers, all parties use $\mathcal{F}_{\text{fan-out-semi}}$ to evaluate fan-out gates (see Section 4.4);
- for each output sharing, all parties use $\mathcal{F}_{\text{permute-semi}}$ to perform the permutation associated with this sharing (see Section 4.2) to achieve the non-collision property (see Section 4.3);
- to prepare each input sharing for the next layer, all parties use $\mathcal{F}_{\text{select-semi}}$ to choose the secrets it wants from the output sharings from previous layers (see Section 4.2), and then use $\mathcal{F}_{\text{permute-semi}}$ to permute the secrets to the correct order (see Section 4.2).

The ideal functionality $\mathcal{F}_{\text{main-semi}}$ appears in Functionality 20. The main protocol is introduced in Protocol 21.

Lemma 9. *Protocol* MAIN-SEMI *securely computes $\mathcal{F}_{\textit{main-semi}}$ in the ($\mathcal{F}_{\textit{input-semi}}, \mathcal{F}_{\textit{fan-out-semi}}, \mathcal{F}_{\textit{permute-semi}}, \mathcal{F}_{\textit{select-semi}}, \mathcal{F}_{\textit{mult-semi}}, \mathcal{F}_{\textit{output-semi}}$)-hybrid model against a semi-honest adversary who controls $t' = t - k + 1$ parties.*

Protocol 19: Preprocess

1. Let C denote the circuit. All parties transform C to C'.

2. Recall that in C', gates that have the same type in each layer are divided into groups of k. During the computation phase, a batch of k wire values are stored in a single packed Shamir sharing. All parties determine how the wire values should be packed as follows:

   - For the input layer, for each group of k input gates belonging to the same client, the values of the output wires will be stored in a single packed Shamir sharing.
   - For each fan-out gate in the input layer, suppose it takes the wires $(w_1, w_2, \ldots, w_k)$ and the vector $(n_1, n_2, \ldots, n_k)$ as input. Recall that in C', $(w_1, w_2, \ldots, w_k)$ are the output wires of a group of gates, n_i is the number of times we need to make copies of w_i, and $n_1+n_2+\ldots+n_k$ is a multiple of k. All parties follow the procedure in $\mathcal{F}_{\text{fan-out-semi}}$ to determine how the output wires should be packed:

     (a) Each party initiates an empty list L. From $i = 1$ to k, each party inserts n_i times of w_i into L.

     (b) Let $m = \frac{n_1+n_2+\ldots+n_k}{k}$. From $i = 1$ to m, the i-th output packed Shamir sharing will contain the values of wires $L((i-1)\cdot k+1), L((i-1)\cdot k+2), \ldots, L(i\cdot k)$.
   - For all intermediate layers, for each group of k multiplication gates or addition gates,
     - the values of the first input wires of these gates will be stored in a single packed Shamir sharing,
     - the values of the second input wires of these gates will be stored in a single packed Shamir sharing,
     - the values of the output wires of these gates will be stored in a single packed Shamir sharing.
   - For each fan-out gate in all intermediate layers, the wire values are packed in the same way as that for each fan-out gate in the input layer.
   - For the output layer, for each group of k output gates belonging to the same client, the values of the input wires will be stored in a single packed Shamir sharing.

3. 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 $1, 2, \ldots, N$.

4. 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 4. Then, all parties associate p_i with the i-th output sharing.

Proof. We first show that Main-semi correctly compute $\mathcal{F}_{\text{main-semi}}$. In the first step, all parties locally run Preprocess. Let C' denote the circuit after transformation. By Theorem 6, it is sufficient to show that all parties correctly compute C'. In Preprocess, all parties determine how the wire values of C' should be stored in packed Shamir sharings. It is sufficient to show that in Main-semi, all parties can obtain correct packed Shamir sharings as they determine in Preprocess.

Functionality 20: $\mathcal{F}_{\text{main-semi}}$

1. $\mathcal{F}_{\text{main-semi}}$ receives the input from all clients. Let x denote the input.
2. $\mathcal{F}_{\text{main-semi}}$ computes $C(x)$ and distributes the output to all clients.

In Preprocess, for the input layer, for each group of k input gates belonging to the same client, the values of the output wires will be stored in a single packed Shamir sharing. In Step 2.(a) of Main-semi, each Client_i ($i \geq 1$) uses $\mathcal{F}_{\text{input-semi}}$ to shares its input. By the correctness of $\mathcal{F}_{\text{input-semi}}$, all parties obtain a packed Shamir sharing of the input of Client_i for each group of k input gates belonging to Client_i. For Client_0, the input values $\boldsymbol{x}^{(0)}$ are constant and known to all parties. Therefore, $[\boldsymbol{x}^{(0)}]$ can be obtained by following the approach in Section 3.2.

In Preprocess, for each fan-out gate in the input layer, all parties follow the same way as that in $\mathcal{F}_{\text{fan-out-semi}}$ to determine the values stored in each output sharing. In Step 2.(b) of Main-semi, all parties invoke $\mathcal{F}_{\text{fan-out-semi}}$ to evaluate each fan-out gate. By the correctness of $\mathcal{F}_{\text{fan-out-semi}}$, all parties obtain correct output sharings for each fan-out gate.

In Preprocess, each output sharing of the input layer and all intermediate layers is associated with a permutation such that after permuting each output sharing using the permutation associated with it, for each input sharing of the output layer and all intermediate layers, the secrets of this sharing come from different positions in the output sharings from previous layers (i.e., the non-collision property). In Step 2.(c) of Main-semi, for each output sharing of the input layer, all parties invoke $\mathcal{F}_{\text{permute-semi}}$ to perform the permutation associated with this sharing. By the correctness of $\mathcal{F}_{\text{permute-semi}}$, all parties obtain a packed Shamir sharing containing the permuted secrets.

From now on, we assume that the secrets of the output sharings of previous layers have been permuted using the permutations associated with these sharings. For each intermediate layer, in Step 3.(a), we prepare the input sharings for this layer. According to the non-collision property, for each input sharing $[\boldsymbol{x}]$ of this layer we want to prepare, the secrets come from different positions in the output sharings of previous layers. Let $[\boldsymbol{x}^{(i)}]$ denote the output sharing from previous layers which contains the i-th secret x_i, and let q_i denote the position of x_i in $[\boldsymbol{x}^{(i)}]$. Then, $q_1, q_2, \ldots, q_k$ is a permutation of $(1, 2, \ldots, k)$. Let $q(\cdot)$ be a permutation over $\{1, 2, \ldots, k\}$ such that $q(i) = q_i$. All parties invoke $\mathcal{F}_{\text{select-semi}}$ on $[\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(k)}]$ and the permutation q. Let $[\boldsymbol{x}']$ denote the output of $\mathcal{F}_{\text{select-semi}}$. By the correctness of $\mathcal{F}_{\text{select-semi}}$, the $q(i)$-th secret $x'_{q(i)}$ is $x^{(i)}_{q(i)} = x_i$. Then $\boldsymbol{x}$ can be obtained by performing q on $\boldsymbol{x}'$. To see this, let $\tilde{\boldsymbol{x}}$ denote the vector after performing q on $\boldsymbol{x}'$. Then $\tilde{x}_i = x'_{q(i)} = x_i$. Therefore, $\tilde{\boldsymbol{x}} = \boldsymbol{x}$. All parties invoke $\mathcal{F}_{\text{permute-semi}}$ with input $[\boldsymbol{x}']$ and q to obtain $[\boldsymbol{x}]$. By the correctness of $\mathcal{F}_{\text{permute-semi}}$, all parties can obtain the correct input sharing $[\boldsymbol{x}]$.

In Preprocess, for all intermediate layers, for each group of k multiplication gates or addition gates, the values of the first input wires of these gates will be stored in a single sharing, the values of the second wires of these gates will be stored in a single sharing, and the values of the output wires will be stored in a single sharing. In the last step, we have shown that all parties in Main-semi can prepare correct input sharings for each intermediate layer, which includes the input sharings for each group of k multiplication gates or addition gates. As for each group of multiplication gates, all parties invoke $\mathcal{F}_{\text{mult-semi}}$ to obtain the output sharing. By the correctness of $\mathcal{F}_{\text{mult-semi}}$, all parties can obtain the correct output sharing for each group of multiplication gates. As for each group of addition gates, by the linear homomorphism of the packed Shamir sharing, all parties can obtain the correct output sharing for each group of addition gates.

In Preprocess, for each fan-out gate in all intermediate layers, all parties follow the same way as that in $\mathcal{F}_{\text{fan-out-semi}}$ to determine the values stored in each output sharing. In Step 3.(c) of Main-semi, all parties invoke $\mathcal{F}_{\text{fan-out-semi}}$ to evaluate each fan-out gate. By the correctness of $\mathcal{F}_{\text{fan-out-semi}}$, all parties obtain correct output sharings for each fan-out gate.

Protocol 21: MAIN-SEMI

1. **Circuit Transformation Phase**. Let C denote the evaluated circuit. All parties preprocess the circuit by running the PREPROCESS protocol. Let C' denote the circuit after transformation.

2. **Input Phase**. Let $\mathsf{Client}_1, \mathsf{Client}_2, \ldots, \mathsf{Client}_c$ denote the clients who provide inputs, and Client_0 denote the virtual client who provides constants. Recall that Client_0 is a virtual client we add during the transformation of C. The input of Client_0 only contains constant values which are known to all parties. We refer the readers to Section 5.1 for more details.

   (a) Input Secret-sharing Phase: For every group of k input gates of Client_i ($i \geq 1$), Client_i invokes $\mathcal{F}_{\text{input-semi}}$ to share its inputs $\boldsymbol{x}^{(i)}$ to the parties. For every group of k input gates of Client_0, the inputs $\boldsymbol{x}^{(0)}$ are constant values and known to all parties. All parties transform $\boldsymbol{x}^{(0)}$ to a degree-t packed Shamir sharing $[\boldsymbol{x}^{(0)}]$ by following the approach in Section 3.2.

   (b) Handling Fan-out Gates: For the output sharing $[\boldsymbol{x}]$ of each group of input gates, let n_i denote the number of times that the i-th secret of $\boldsymbol{x}$ is used in later layers. All parties invoke $\mathcal{F}_{\text{fan-out-semi}}$ with input $[\boldsymbol{x}]$ and $(n_1, n_2, \ldots, n_k)$.

   (c) Achieving Non-Collision Property for the Next Layers: For each output sharing $[\boldsymbol{y}]$ of the input layer, let p denote the permutation associated with it. All parties invoke $\mathcal{F}_{\text{permute-semi}}$ with input $[\boldsymbol{y}]$ and p.

3. **Evaluation Phase**. All parties evaluate the circuit layer by layer as follows:

   (a) Permute Input Sharings from Previous Layers: For each input sharing $[\boldsymbol{x}]$, let $[\boldsymbol{x}^{(i)}]$ denote the output sharing from previous layers which contains the i-th secret x_i, and let q_i denote the position of x_i in $[\boldsymbol{x}^{(i)}]$. According to the non-collision property, $q_1, q_2, \ldots, q_k$ is a permutation of $(1, 2, \ldots, k)$. Let $q(\cdot)$ be a permutation over $\{1, 2, \ldots, k\}$ such that $q(i) = q_i$. All parties invoke $\mathcal{F}_{\text{select-semi}}$ on $[\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(k)}]$ and the permutation q. Let $[\boldsymbol{x}']$ denote the output of $\mathcal{F}_{\text{select-semi}}$. Then, all parties invoke $\mathcal{F}_{\text{permute-semi}}$ with input $[\boldsymbol{x}']$ and q to obtain $[\boldsymbol{x}]$.

   (b) Evaluating Multiplication Gates and Addition Gates: For each group of multiplication gates with input sharings $[\boldsymbol{x}], [\boldsymbol{y}]$, all parties invoke $\mathcal{F}_{\text{mult-semi}}$ with input $[\boldsymbol{x}], [\boldsymbol{y}]$. For each group of addition gates with input sharings $[\boldsymbol{x}], [\boldsymbol{y}]$, all parties locally compute $[\boldsymbol{x} + \boldsymbol{y}] = [\boldsymbol{x}] + [\boldsymbol{y}]$.

   (c) Handling Fan-out Gates: For the output sharing $[\boldsymbol{x}]$ of each group of multiplication gates or addition gates, all parties follow the same step as Step 2.(b) to handle fan-out gates.

   (d) Achieving Non-Collision Property: Follow Step 2.(c).

4. **Output Phase**.

   (a) Permute Input Sharings from Previous Layers: For each input sharing $[\boldsymbol{x}]$, all parties follow the same step as Step 3.(a) to prepare $[\boldsymbol{x}]$.

   (b) Reconstruct the Output: For each group of output gates belonging to Client_i ($i \geq 1$), let $[\boldsymbol{x}]$ denote the input sharing. All parties invoke $\mathcal{F}_{\text{output-semi}}$ with input $[\boldsymbol{x}]$ to let Client_i learn the result $\boldsymbol{x}$.

In Step 3.(d) of MAIN-SEMI, all parties permute the secrets of each output sharings using the permutation associated with this sharing to achieve the non-collision property.

For the output layer, in Step 4.(a), all parties prepare the input sharings of this layer. Similarly to that for each intermediate layer, all parties can obtain correct input sharings relying on the non-collision property, $\mathcal{F}_{\text{select-semi}}$, and $\mathcal{F}_{\text{permute-semi}}$.

Finally, in Step 4.(b), all parties invoke $\mathcal{F}_{\text{output-semi}}$ to reconstruct the output to each client Client_i ($i \geq 1$). Thus, the correctness holds.

As for security, note that we can construct a simulator $\mathcal{S}$ which simply emulates the functionalities $\mathcal{F}_{\text{input-semi}}, \mathcal{F}_{\text{fan-out-semi}}, \mathcal{F}_{\text{permute-semi}}, \mathcal{F}_{\text{select-semi}}, \mathcal{F}_{\text{mult-semi}}$ during the computation. $\mathcal{S}$ learns the inputs of corrupted clients in $\mathcal{F}_{\text{input-semi}}$ and then passes them to $\mathcal{F}_{\text{main-semi}}$. In the meantime, $\mathcal{S}$ learns the shares of each degree-t packed Shamir sharing held by corrupted parties when simulating these functionalities. Therefore, $\mathcal{S}$ can compute the shares of each sharing associated with the output gates belonging to corrupted clients by following the protocol. After receiving the outputs of corrupted client from $\mathcal{F}_{\text{main-semi}}$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{output-semi}}$ using the shares of corrupted parties it computes and the results received from $\mathcal{F}_{\text{main-semi}}$. Note that, during the computation, the adversary only receives messages from honest parties or ideal functionalities when invoking $\mathcal{F}_{\text{output-semi}}$, which includes the shares of corrupted parties and the final results. Since the shares of corrupted parties are computed by $\mathcal{S}$ following the protocol, the distribution of these shares is the same in both the ideal world and the real world. The final results are also the same in both worlds according to the correctness of MAIN-SEMI. Also, the output of each honest client is identical in both worlds according to the correctness of MAIN-SEMI. Thus, MAIN-SEMI securely computes $\mathcal{F}_{\text{main-semi}}$. □

Analysis of the Communication Complexity of Main-semi. Note that in MAIN-SEMI, interaction is needed only when invoking the functionalities $\mathcal{F}_{\text{input-semi}}, \mathcal{F}_{\text{fan-out-semi}}, \mathcal{F}_{\text{permute-semi}}, \mathcal{F}_{\text{select-semi}}, \mathcal{F}_{\text{mult-semi}}, \mathcal{F}_{\text{output-semi}}$.

For $\mathcal{F}_{\text{input-semi}}, \mathcal{F}_{\text{mult-semi}}, \mathcal{F}_{\text{output-semi}}$, they are used to evaluate input gates, multiplication gates, and output gates in C'. Since each functionality evaluates k gates of the same type each time, the total number of invocations of these functionalities is bounded by $|C'|/k$. Recall that for $\mathcal{F}_{\text{input-semi}}, \mathcal{F}_{\text{output-semi}}$, the implementations INPUT, OUTPUT have communication complexity of $O(n)$ field elements. For $\mathcal{F}_{\text{mult-semi}}$, for all $m \geq 1$, m invocations of the implementation MULT have communication complexity of $O(m \cdot n + n^3 \cdot \kappa)$ field elements. Therefore, the total communication complexity of $\mathcal{F}_{\text{input-semi}}, \mathcal{F}_{\text{mult-semi}}, \mathcal{F}_{\text{output-semi}}$ is bounded by $O(|C'| \cdot n/k + n^3 \cdot \kappa)$ field elements.

For $\mathcal{F}_{\text{fan-out-semi}}$, recall that the communication complexity of using the implementation FAN-OUT to generate m output sharings is $O(m \cdot n + n^5 \cdot \kappa)$ field elements. Note that we invoke $\mathcal{F}_{\text{fan-out-semi}}$ for the input layer and all intermediate layers right after evaluating input gates, multiplication gates, and addition gates. The output sharings of $\mathcal{F}_{\text{fan-out-semi}}$ are the output sharings of these layers. Since the number of output wires of the input layer and all intermediate layers is bounded by $|C'|$, the total number of output sharings is bounded by $|C'|/k$. Thus, the total communication complexity of $\mathcal{F}_{\text{fan-out-semi}}$ is bounded by $O(|C'| \cdot n/k + n^5 \cdot \kappa)$ field elements.

For $\mathcal{F}_{\text{permute-semi}}$, it is invoked once for each of output sharings and input sharings. Recall that for all $m \geq 1$, m invocations of the implementation PERMUTE have communication complexity $O(m \cdot n + n^5 \cdot \kappa)$ field elements. Therefore, the total communication complexity of $\mathcal{F}_{\text{permute-semi}}$ is $O(|C'| \cdot n/k + n^5 \cdot \kappa)$ field elements.

For $\mathcal{F}_{\text{select-semi}}$, it is invoked once to prepare each input sharing for the output layer and all intermediate layers. Since for all $m \geq 1$, m invocations of the implementation SELECT have communication complexity $O(m \cdot n + n^3 \cdot \kappa)$ field elements, the total communication complexity of $\mathcal{F}_{\text{select-semi}}$ is bounded by $O(|C'| \cdot n/k + n^3 \cdot \kappa)$ field elements.

Plugging in $|C'| = O(|C| + k \cdot (c + \mathsf{Depth}))$, the communication complexity of MAIN-SEMI is

$$O(|C| \cdot n/k + n \cdot (c + \mathsf{Depth}) + n^5 \cdot \kappa)$$

field elements.

Theorem 7. *In the client-server model, let c denote the number of clients, and $n = 2t+1$ denote the number of parties (servers). Let κ denote the security parameter, and $\mathbb{F}$ denote a finite field. For an arithmetic circuit C over $\mathbb{F}$ and for all $1 \leq k \leq t$, there exists an information-theoretic MPC protocol which securely computes the arithmetic circuit C in the presence of a semi-honest adversary controlling up to c clients and $t-k+1$ parties. The communication complexity of this protocol is $O(|C| \cdot n/k + n \cdot (c + \mathsf{Depth}) + n^5 \cdot \kappa)$ elements in $\mathbb{F}$.*

6 Towards Security (with Abort) Against a Fully Malicious Adversary

In this section, we discuss how to achieve malicious security with abort. We observe that most of protocols described in Section 4 have already achieved perfect privacy *against a fully malicious adversary*, namely the executions of these protocols do not leak any information to the adversary. Also, the deviation of a fully malicious adversary can be reduced to the following two kinds of attacks:

- An adversary can distribute an inconsistent degree-t packed Shamir sharing.
- An adversary can add additive errors to the secrets of the output sharing.

To achieve malicious security, our idea is to run our semi-honest protocol described in Section 5 before the output phase, check whether the above two kinds of attacks are launched by the adversary, and finally reconstruct the output.

We first prove the security of our protocols described in Section 4 against a fully malicious adversary. Recall that we are in the client-server model where there are c clients and $n = 2t+1$ parties (servers). Recall that $1 \leq k \leq t$ is an integer. An adversary is allowed to corrupt $t' = t - k + 1$ parties. We will use the degree-t packed Shamir sharing scheme, which can store k secrets within one sharing. Recall that $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties.

Let $\mathcal{H}_{\mathcal{H}} \subset \mathcal{H}$ be a fixed subset of size $t+1$, and $\mathcal{H}_{\mathcal{C}} = \mathcal{H} \backslash \mathcal{H}_{\mathcal{H}}$. Note that the shares of parties in $\mathcal{H}_{\mathcal{H}}$ can fully determine a degree-t packed Shamir sharing $[\boldsymbol{x}]$. For a degree-t packed Shamir sharing $[\boldsymbol{x}]$ held by all parties,

- we use the notation $[\boldsymbol{x}]_{\mathcal{H}}$ to denote the degree-t packed Shamir sharing determined by the shares of parties in $\mathcal{H}_{\mathcal{H}}$;
- and we use $[\boldsymbol{x}]_{\mathcal{C}}$ to denote the sharing where the shares of parties in $\mathcal{H}$ are identical to the shares of $[\boldsymbol{x}]$ held by parties in $\mathcal{H}$, and the shares of parties in $\mathcal{C}$ are identical to the shares of $[\boldsymbol{x}]_{\mathcal{H}}$ held by parties in $\mathcal{C}$.

Note that $[\boldsymbol{x}]_{\mathcal{H}}$ is always a valid degree-t packed Shamir sharing, while $[\boldsymbol{x}]_{\mathcal{C}}$ can be inconsistent. We will maintain the invariant that the adversary learns the shares of $[\boldsymbol{x}]_{\mathcal{H}}$ held by corrupted parties (i.e., the shares corrupted parties should hold), and the difference $\boldsymbol{\Delta} = [\boldsymbol{x}]_{\mathcal{C}} - [\boldsymbol{x}]_{\mathcal{H}}$, which describes the inconsistency of $[\boldsymbol{x}]$ due to the deviation of corrupted parties. Note that $\Delta_i \neq 0$ iff $P_i \in \mathcal{H}_{\mathcal{C}}$. The secrets of $[\boldsymbol{x}]$ are defined to be the secrets of $[\boldsymbol{x}]_{\mathcal{H}}$.

Note that for honest parties, the shares of $[\boldsymbol{x}]_{\mathcal{C}}$ are identical to the shares of $[\boldsymbol{x}]$. The only difference between $[\boldsymbol{x}]$ and $[\boldsymbol{x}]_{\mathcal{C}}$ is that we specify the shares of corrupted parties in $[\boldsymbol{x}]_{\mathcal{C}}$ to be the shares they should hold. Since we will maintain the invariant that the adversary learns the shares that corrupted parties should hold, we can assume that corrupted parties hold the correct shares while they may use incorrect values during the computation. We will simply use $[\boldsymbol{x}]$ to represent $[\boldsymbol{x}]_{\mathcal{C}}$. Note that, both $[\boldsymbol{x}]_{\mathcal{H}}$ and $[\boldsymbol{x}]$ are determined by the shares of all honest parties.

6.1 Security of Input, Output, and Mult

Security of Input and Output. We first describe the functionalities $\mathcal{F}_{\text{input-mal}}$ and $\mathcal{F}_{\text{output-mal}}$. For $\mathcal{F}_{\text{input-mal}}$, the functionality allows the adversary to further specify an additive error $\boldsymbol{\Delta}$ which is added to the output sharing. For $\mathcal{F}_{\text{output-mal}}$, the functionality further sends the shares of corrupted parties and the additive error $\boldsymbol{\Delta}$ of the input sharing to the adversary, and it allows the adversary to abort the protocol at any point. For $\mathcal{F}_{\text{input-mal}}$, we show that INPUT securely computes $\mathcal{F}_{\text{input-mal}}$ against a fully malicious adversary. For $\mathcal{F}_{\text{output-mal}}$, we change OUTPUT by requiring each client who receives an inconsistent sharing in Step 2 to abort the protocol. Then we show that OUTPUT securely computes $\mathcal{F}_{\text{output-mal}}$ against a fully malicious adversary. We mark the changes in blue for Functionalities $\mathcal{F}_{\text{input-mal}}, \mathcal{F}_{\text{output-mal}}$ and Protocol OUTPUT-MAL compared with those in the semi-honest setting.

Functionality 22: $\mathcal{F}_{\text{input-mal}}$

1. Suppose $\boldsymbol{x} \in \mathbb{F}^k$ is the input associated with the input gate which belongs to the Client. $\mathcal{F}_{\text{input-mal}}$ receives the input $\boldsymbol{x}$ from the Client.

2. $\mathcal{F}_{\text{input-mal}}$ receives from the adversary a set of shares $\{s_i\}_{i \in \mathcal{C}}$. $\mathcal{F}_{\text{input-mal}}$ samples a random degree-t packed Shamir sharing $[\boldsymbol{x}]_{\mathcal{H}}$ such that for all $P_i \in \mathcal{C}$, the i-th share of $[\boldsymbol{x}]_{\mathcal{H}}$ is s_i.

3. $\mathcal{F}_{\text{input-mal}}$ receives from the adversary a vector $\boldsymbol{\Delta} \in \mathbb{F}^n$ such that for all $P_i \notin \mathcal{H}_{\mathcal{C}}$, $\Delta_i = 0$. $\mathcal{F}_{\text{input-mal}}$ computes $[\boldsymbol{x}] = [\boldsymbol{x}]_{\mathcal{H}} + \boldsymbol{\Delta}$.

4. $\mathcal{F}_{\text{input-mal}}$ distributes the shares of $[\boldsymbol{x}]$ to honest parties.

Functionality 23: $\mathcal{F}_{\text{output-mal}}$

1. Suppose $[\boldsymbol{x}]$ is the sharing associated with the output gate which belongs to the Client. $\mathcal{F}_{\text{output-mal}}$ receives the shares of $[\boldsymbol{x}]$ from honest parties.

2. $\mathcal{F}_{\text{output-mal}}$ recovers the whole sharings $[\boldsymbol{x}]_{\mathcal{H}}$ and $[\boldsymbol{x}]$. Let $\boldsymbol{x}$ denote the secrets of $[\boldsymbol{x}]_{\mathcal{H}}$. $\mathcal{F}_{\text{output-mal}}$ computes $\boldsymbol{\Delta} = [\boldsymbol{x}] - [\boldsymbol{x}]_{\mathcal{H}}$ and sends $\boldsymbol{\Delta}$ and the shares of $[\boldsymbol{x}]_{\mathcal{H}}$ of corrupted parties to the adversary.

3. $\mathcal{F}_{\text{output-mal}}$ reconstructs $\boldsymbol{x}$. Depending on whether the Client is honest or not, there are two cases:

   - If the Client is corrupted, $\mathcal{F}_{\text{output-mal}}$ sends $\boldsymbol{x}$ to the adversary. If the adversary replies `abort`, $\mathcal{F}_{\text{output-mal}}$ sends `abort` to all honest parties.
   - If the Client is honest, $\mathcal{F}_{\text{output-mal}}$ asks the adversary whether it should continue. If the adversary replies `abort`, $\mathcal{F}_{\text{output-mal}}$ sends `abort` to the Client and all honest parties. If the adversary replies `continue`, $\mathcal{F}_{\text{output-mal}}$ sends $\boldsymbol{x}$ to the Client.

Protocol 24: OUTPUT-MAL

1. Suppose $[\boldsymbol{x}]$ is the sharing associated with the output gate which belongs to the Client. All parties send their shares of $[\boldsymbol{x}]$ to the Client.

2. If the Client receives an inconsistent sharing $[\boldsymbol{x}]$, the Client aborts. Otherwise, the Client reconstructs the result $\boldsymbol{x}$ from the shares of $[\boldsymbol{x}]$.

Lemma 10. *Protocol* INPUT *(see Protocol 5) securely computes $\mathcal{F}_{\text{input-mal}}$ against a fully malicious adversary who controls $t' = t - k + 1$ parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that $\mathcal{H}_{\mathcal{H}} \subset \mathcal{H}$ is a fixed subset of size $t + 1$.

If the Client is corrupted, $\mathcal{S}$ receives the shares of honest parties from the Client. Then $\mathcal{S}$ recovers the whole sharings $[\boldsymbol{x}]_{\mathcal{H}}$ and $[\boldsymbol{x}]$. $\mathcal{S}$ computes the difference $\boldsymbol{\Delta} = [\boldsymbol{x}] - [\boldsymbol{x}]_{\mathcal{H}}$. $\mathcal{S}$ sends the secrets $\boldsymbol{x}$ to $\mathcal{F}_{\text{input-mal}}$ on behalf of the Client. $\mathcal{S}$ also sends the shares of $[\boldsymbol{x}]_{\mathcal{H}}$ of corrupted parties and $\boldsymbol{\Delta}$ to $\mathcal{F}_{\text{input-mal}}$. Note that a degree-$t$ packed Shamir sharing $[\boldsymbol{x}]_{\mathcal{H}}$ is determined by the shares of corrupted parties and the secrets, which are $t'+k=t+1$ shares and secrets. Therefore, the sharing $[\boldsymbol{x}]_{\mathcal{H}}$ generated by $\mathcal{F}_{\text{input-mal}}$ is identical to $[\boldsymbol{x}]_{\mathcal{H}}$ reconstructed by $\mathcal{S}$. Furthermore, using $\boldsymbol{\Delta}$, $\mathcal{F}_{\text{input-mal}}$ can compute the same sharing $[\boldsymbol{x}]$ as that computed by $\mathcal{S}$. Recall that for $[\boldsymbol{x}]$, the shares of honest parties are identical to the shares received from the dealer Client. Thus, the shares of honest parties distributed by $\mathcal{F}_{\text{input-mal}}$ in the ideal world are identical to the shares of honest parties distributed by the Client in the real world.

If the Client is honest, $\mathcal{S}$ generates t' random elements in $\mathbb{F}$ as the shares of $[\boldsymbol{x}]$ held by corrupted parties and distributes them to corrupted parties on behalf of the Client. Note that these shares have the same distribution as the shares generated by the Client in the real world. $\mathcal{S}$ sends the shares of $[\boldsymbol{x}]$ of corrupted parties and an all-0 vector $\boldsymbol{\Delta}$ to $\mathcal{F}_{\text{input-mal}}$. Note that when the Client is honest, the sharing distributed by the Client is consistent, i.e., $[\boldsymbol{x}]_{\mathcal{H}} = [\boldsymbol{x}]$. Since the whole sharing $[\boldsymbol{x}]_{\mathcal{H}}$ is determined by the shares of corrupted parties and the secrets, and the shares of corrupted parties in both worlds have the same distribution, the distributions of $[\boldsymbol{x}]_{\mathcal{H}}$ in both worlds are identical. Since $[\boldsymbol{x}]_{\mathcal{H}} = [\boldsymbol{x}]$, the distributions of the shares held by honest parties in both worlds are identical. □

Lemma 11. *Protocol* OUTPUT-MAL *(see Protocol 24) securely computes $\mathcal{F}_{output\text{-}mal}$ against a fully malicious adversary who controls $t' = t-k+1$ parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that $\mathcal{H}_{\mathcal{H}} \subset \mathcal{H}$ is a fixed subset of size $t+1$.

$\mathcal{S}$ first receives from $\mathcal{F}_{\text{output-mal}}$ the shares of $[\boldsymbol{x}]_{\mathcal{H}}$ held by corrupted parties and the difference $\boldsymbol{\Delta} = [\boldsymbol{x}] - [\boldsymbol{x}]_{\mathcal{H}}$.

If the Client is corrupted, $\mathcal{S}$ also receives the secrets $\boldsymbol{x}$ from $\mathcal{F}_{\text{output-mal}}$. Then $\mathcal{S}$ recovers the whole sharing $[\boldsymbol{x}]_{\mathcal{H}}$ by using the secrets $\boldsymbol{x}$ and the shares of $[\boldsymbol{x}]_{\mathcal{H}}$ held by corrupted parties. $\mathcal{S}$ computes $[\boldsymbol{x}]$ by $[\boldsymbol{x}] = [\boldsymbol{x}]_{\mathcal{H}} + \boldsymbol{\Delta}$ and sends the shares of $[\boldsymbol{x}]$ of honest parties to the Client. Note that a degree-t packed Shamir sharing is determined by the shares of corrupted parties and the secrets, which are $t'+k=t+1$ shares and secrets. Therefore, the sharing $[\boldsymbol{x}]_{\mathcal{H}}$ computed by $\mathcal{S}$ is identical to the sharing $[\boldsymbol{x}]_{\mathcal{H}}$ computed by $\mathcal{F}_{\text{output-mal}}$. Using $\boldsymbol{\Delta}$ received from $\mathcal{F}_{\text{output-mal}}$, $\mathcal{S}$ computes the sharing $[\boldsymbol{x}]$, where the shares of $[\boldsymbol{x}]$ of honest parties are identical to the shares $\mathcal{F}_{\text{output-mal}}$ received from honest parties. Therefore the distribution of the shares of honest parties computed by $\mathcal{S}$ in the ideal world is identical to the distribution of the shares held by honest parties in the real world. If the Client aborts, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{output-mal}}$.

If the Client is honest, $\mathcal{S}$ receives from corrupted parties their shares of $[\boldsymbol{x}]$. Then $\mathcal{S}$ checks whether the shares of $[\boldsymbol{x}]$ $\mathcal{S}$ received from corrupted parties are the same as the shares of $[\boldsymbol{x}]_{\mathcal{H}}$ $\mathcal{S}$ received from $\mathcal{F}_{\text{output-mal}}$, and whether $\boldsymbol{\Delta} = 0$. If both checks pass, $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\text{output-mal}}$. Otherwise, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{output-mal}}$. Note that in the real world, the Client does not abort if and only if $[\boldsymbol{x}]$ is consistent, i.e., $[\boldsymbol{x}] = [\boldsymbol{x}]_{\mathcal{H}}$. If either check fails, the Client does not receive a consistent sharing and will abort in the real world. Otherwise, the Client can reconstruct the correct output $\boldsymbol{x}$. Therefore, the output of the Client is identical in both worlds. □

Security of Mult. Suppose the input sharings of MULT are denoted by $[\boldsymbol{x}], [\boldsymbol{y}]$, and $[\boldsymbol{x}]_{\mathcal{H}}, [\boldsymbol{y}]_{\mathcal{H}}$ are defined accordingly. Ideally, we want the ideal functionality to

1. compute $\boldsymbol{x}, \boldsymbol{y}$ from $[\boldsymbol{x}]_{\mathcal{H}}, [\boldsymbol{y}]_{\mathcal{H}}$;

2. compute the multiplication result $\boldsymbol{z} := \boldsymbol{x} * \boldsymbol{y}$, where $*$ denotes the coordinate-wise multiplication;

3. receive additive errors $\boldsymbol{d}$ to the secrets $\boldsymbol{z}$ and additive errors $\boldsymbol{\Delta}$ to the shares held by parties in $\mathcal{H}_{\mathcal{C}}$;

4. generate a degree-t packed Shamir sharing $[\boldsymbol{z}+\boldsymbol{d}]_{\mathcal{H}}$ and then add the additive errors $[\boldsymbol{z}+\boldsymbol{d}] := [\boldsymbol{z}+\boldsymbol{d}]_{\mathcal{H}} + \boldsymbol{\Delta}$.

However, there is a subtle issue due to the inconsistency of the input sharings. Recall that in Mult, all parties locally compute $\langle \boldsymbol{z} \rangle = [\boldsymbol{x}] \cdot [\boldsymbol{y}]$ and then reduce the degree of $\langle \boldsymbol{z} \rangle$. Even if corrupted parties honestly use their shares of $[\boldsymbol{x}], [\boldsymbol{y}]$ (i.e., the same shares as those of $[\boldsymbol{x}]_{\mathcal{H}}, [\boldsymbol{y}]_{\mathcal{H}}$) and honestly follow the protocol, all parties can only compute

$$\langle \boldsymbol{z} \rangle = [\boldsymbol{x}] \cdot [\boldsymbol{y}] = [\boldsymbol{x}]_{\mathcal{H}} \cdot [\boldsymbol{y}]_{\mathcal{H}} + \boldsymbol{\Delta}_x * [\boldsymbol{y}]_{\mathcal{H}} + [\boldsymbol{x}]_{\mathcal{H}} * \boldsymbol{\Delta}_y + \boldsymbol{\Delta}_x * \boldsymbol{\Delta}_y,$$

where $\boldsymbol{\Delta}_x = [\boldsymbol{x}] - [\boldsymbol{x}]_{\mathcal{H}}$ and $\boldsymbol{\Delta}_y = [\boldsymbol{y}] - [\boldsymbol{y}]_{\mathcal{H}}$. Unfortunately, the cross term $\boldsymbol{\Delta}_x * [\boldsymbol{y}]_{\mathcal{H}} + [\boldsymbol{x}]_{\mathcal{H}} * \boldsymbol{\Delta}_y$ adds errors which are related to the inputs $\boldsymbol{x}, \boldsymbol{y}$ to the secrets $\boldsymbol{x} * \boldsymbol{y}$. It means that we cannot reduce this kind of errors to additive errors chosen by the adversary.

To solve it, the functionality will compute $\boldsymbol{z}$ using $\langle \boldsymbol{z} \rangle = [\boldsymbol{x}] \cdot [\boldsymbol{y}]$ instead of using the secrets $\boldsymbol{x}, \boldsymbol{y}$. It means that $\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{y}$ holds only if $\boldsymbol{\Delta}_x = \boldsymbol{\Delta}_y = \boldsymbol{0}$. To ensure the correctness, all parties will verify the consistency of all degree-t packed Shamir sharings at the end of the protocol. The description of $\mathcal{F}_{\text{mult-mal}}$ appears in Functionality 25. We mark the changes in blue for Functionality $\mathcal{F}_{\text{mult-mal}}$ compared with that in the semi-honest setting.

Functionality 25: $\mathcal{F}_{\text{mult-mal}}$

1. Suppose $[\boldsymbol{x}], [\boldsymbol{y}]$ are the input degree-t packed Shamir sharings. $\mathcal{F}_{\text{mult-mal}}$ receives the shares of $[\boldsymbol{x}], [\boldsymbol{y}]$ from honest parties.
2. $\mathcal{F}_{\text{mult-mal}}$ recovers the whole sharings $[\boldsymbol{x}]_{\mathcal{H}}, [\boldsymbol{x}]$ and $[\boldsymbol{y}]_{\mathcal{H}}, [\boldsymbol{y}]$. $\mathcal{F}_{\text{mult-mal}}$ computes $\boldsymbol{\Delta}_x = [\boldsymbol{x}] - [\boldsymbol{x}]_{\mathcal{H}}$ and $\boldsymbol{\Delta}_y = [\boldsymbol{y}] - [\boldsymbol{y}]_{\mathcal{H}}$. Then $\mathcal{F}_{\text{mult-mal}}$ sends the shares of $[\boldsymbol{x}]_{\mathcal{H}}, [\boldsymbol{y}]_{\mathcal{H}}$ of corrupted parties and $\boldsymbol{\Delta}_x, \boldsymbol{\Delta}_y$ to the adversary.
3. $\mathcal{F}_{\text{mult-mal}}$ receives from the adversary a vector $\boldsymbol{d} \in \mathbb{F}^k$. $\mathcal{F}_{\text{mult-mal}}$ computes $\langle \boldsymbol{z} \rangle := [\boldsymbol{x}] \cdot [\boldsymbol{y}]$ and reconstructs the secrets $\boldsymbol{z}$. $\mathcal{F}_{\text{mult-mal}}$ computes $\boldsymbol{z} := \boldsymbol{z} + \boldsymbol{d}$.
4. $\mathcal{F}_{\text{mult-mal}}$ receives from the adversary a set of shares $\{s_i\}_{i \in \mathcal{C}}$. $\mathcal{F}_{\text{mult-mal}}$ samples a random degree-t packed Shamir sharing $[\boldsymbol{z}]_{\mathcal{H}}$ such that for all $P_i \in \mathcal{C}$, the i-th share of $[\boldsymbol{z}]_{\mathcal{H}}$ is s_i.
5. $\mathcal{F}_{\text{mult-mal}}$ receives from the adversary a vector $\boldsymbol{\Delta}_z \in \mathbb{F}^n$ such that for all $P_i \notin \mathcal{H}_{\mathcal{C}}$, $(\boldsymbol{\Delta}_z)_i = 0$. $\mathcal{F}_{\text{mult-mal}}$ computes $[\boldsymbol{z}] = [\boldsymbol{z}]_{\mathcal{H}} + \boldsymbol{\Delta}_z$.
6. $\mathcal{F}_{\text{mult-mal}}$ distributes the shares of $[\boldsymbol{z}]$ to honest parties.

Lemma 12. *Protocol* Mult *(see Protocol 8) securely computes $\mathcal{F}_{\text{mult-mal}}$ in the $\mathcal{F}_{\text{rand}}$-hybrid model against a fully malicious adversary who controls $t' = t - k + 1$ parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that $\mathcal{H}_{\mathcal{H}} \subset \mathcal{H}$ is a fixed subset of size $t+1$.

Recall that in Lemma 3, we have shown that, for a pair of two random sharings $([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$, given the shares of $[\boldsymbol{r}], \langle \boldsymbol{r} \rangle$ held by corrupted parties, the shares of $\langle \boldsymbol{r} \rangle$ held by honest parties are uniform.

Simulation for Mult. Now we describe the construction of the simulator $\mathcal{S}$.

- In the beginning, $\mathcal{S}$ receives from $\mathcal{F}_{\text{mult-mal}}$ the shares of $[\boldsymbol{x}]_{\mathcal{H}}, [\boldsymbol{y}]_{\mathcal{H}}$ held by corrupted parties and $\boldsymbol{\Delta}_x, \boldsymbol{\Delta}_y$.
- In Step 2, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ and receives the shares $\{(s_i^{(0)}, s_i^{(1)})\}_{i \in \mathcal{C}}$ from the adversary.

- In Step 3, $\mathcal{S}$ generates a random element in $\mathbb{F}$ for each honest party as its share of $\langle \boldsymbol{e} \rangle$. Since the shares of $\langle \boldsymbol{r} \rangle$ of honest parties in the real world are uniform, the distribution of the shares of $\langle \boldsymbol{e} \rangle$ of honest parties generated by $\mathcal{S}$ is identical to that in the real world. Using the shares of $[\boldsymbol{x}]_{\mathcal{H}}, [\boldsymbol{y}]_{\mathcal{H}}, \langle \boldsymbol{r} \rangle$ held by corrupted parties, $\mathcal{S}$ computes the shares of $\langle \boldsymbol{e} \rangle$ that corrupted parties should hold. Then $\mathcal{S}$ reconstructs the secrets $\boldsymbol{e}$.

- $\mathcal{S}$ honestly follows Step 4 and Step 5. Let $[\boldsymbol{e}']$ denote the sharing distributed by P_1. At the end of Step 5, $\mathcal{S}$ learns the shares of $[\boldsymbol{e}']$ held by honest parties. $\mathcal{S}$ reconstructs the whole sharings $[\boldsymbol{e}']_{\mathcal{H}}, [\boldsymbol{e}']$ and computes the secrets $\boldsymbol{e}'$ of $[\boldsymbol{e}']_{\mathcal{H}}$. $\mathcal{S}$ sets $\boldsymbol{d} = \boldsymbol{e}' - \boldsymbol{e}$ and $\boldsymbol{\Delta}_z = [\boldsymbol{e}'] - [\boldsymbol{e}']_{\mathcal{H}}$.

- In Step 6, $\mathcal{S}$ computes the shares of $[\boldsymbol{z}]_{\mathcal{H}}$ held by corrupted parties by using the shares of $[\boldsymbol{e}']_{\mathcal{H}}$ and $[\boldsymbol{r}]$ held by corrupted parties.

- Finally, $\mathcal{S}$ sends to $\mathcal{F}_{\text{mult-mal}}$ the vector $\boldsymbol{d}$, the shares of $[\boldsymbol{z}]_{\mathcal{H}}$ held by corrupted parties, and the vector $\boldsymbol{\Delta}_z$.

Analyze the Security of Mult. It is clear that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. It is sufficient to show that the shares of honest parties in both worlds have the same distribution.

- In the real world, we have $\langle \boldsymbol{e} \rangle := [\boldsymbol{x}] \cdot [\boldsymbol{y}] + \langle \boldsymbol{r} \rangle$. Let $[\boldsymbol{e}']$ denote the degree-t packed Shamir sharing distributed by P_1. Then, the secrets of the output sharing are $\boldsymbol{e}' - \boldsymbol{r} = (\boldsymbol{e}' - \boldsymbol{e}) + (\boldsymbol{e} - \boldsymbol{r})$. In the ideal world, $\mathcal{F}_{\text{mult-mal}}$ computes the secrets $\boldsymbol{z}$ of $\langle \boldsymbol{z} \rangle := [\boldsymbol{x}] \cdot [\boldsymbol{y}]$, which are equal to $\boldsymbol{e} - \boldsymbol{r}$. The simulator computes $\boldsymbol{d} = \boldsymbol{e}' - \boldsymbol{e}$ as above and sends $\boldsymbol{d}$ to $\mathcal{F}_{\text{mult-mal}}$. The final secrets are set to be $\boldsymbol{z} := \boldsymbol{z} + \boldsymbol{d} = \boldsymbol{e} - \boldsymbol{r} + \boldsymbol{e}' - \boldsymbol{e} = \boldsymbol{e}' - \boldsymbol{r}$. Thus, the secrets $\boldsymbol{z}$ in both worlds have the same distribution.

- In the real world, the output sharing $[\boldsymbol{z}]$ is computed by $[\boldsymbol{e}'] - [\boldsymbol{r}]$. Then we have $[\boldsymbol{z}] = [\boldsymbol{e}'] - [\boldsymbol{r}]$ and $[\boldsymbol{z}]_{\mathcal{H}} = [\boldsymbol{e}']_{\mathcal{H}} - [\boldsymbol{r}]$ since $[\boldsymbol{r}]$ is guaranteed to be consistent according to $\mathcal{F}_{\text{rand}}$. Recall that the shares of $[\boldsymbol{z}]$ that corrupted parties should hold are the shares of $[\boldsymbol{z}]_{\mathcal{H}}$ of corrupted parties. In the ideal world, $\mathcal{S}$ computes the shares of $[\boldsymbol{z}]_{\mathcal{H}}$ of corrupted parties using the shares of $[\boldsymbol{e}']_{\mathcal{H}}$ and $[\boldsymbol{r}]$ of corrupted parties as above. Therefore, the shares of $[\boldsymbol{z}]$ that corrupted parties should hold in both worlds are identical.

- In the real world, the additive errors to the shares of $[\boldsymbol{z}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ are $[\boldsymbol{z}] - [\boldsymbol{z}]_{\mathcal{H}}$. Note that $[\boldsymbol{z}] - [\boldsymbol{z}]_{\mathcal{H}} = [\boldsymbol{e}'] - [\boldsymbol{e}']_{\mathcal{H}}$. Therefore, the additive errors $\boldsymbol{\Delta}_z = [\boldsymbol{e}'] - [\boldsymbol{e}']_{\mathcal{H}}$ computed by $\mathcal{S}$ are identical to the additive errors in the real world.

Note that $[\boldsymbol{z}]$ distributed by $\mathcal{F}_{\text{mult-mal}}$ is determined by the secrets $\boldsymbol{z}$, the shares of corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. The distributions of them in both worlds are identical. Therefore, the shares of honest parties in both worlds have the same distribution. □

6.2 Security of Select, Rand-Perm, and Permute

Security of Select. Recall that the functionality of SELECT is to select k secrets in different positions from k different input degree-t packed Shamir sharings and output a single degree-t packed Shamir sharing which contains the chosen secrets. More concretely, let $[\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(k)}]$ denote the input sharings, and p be a permutation over $\{1, 2, \ldots, k\}$. The goal is to generate a degree-t packed Shamir sharing of $\boldsymbol{y}$ such that for all $i \in \{1, 2, \ldots, k\}$, $y_{p(i)} = x^{(i)}_{p(i)}$, i.e., choosing the $p(i)$-th secret of $[\boldsymbol{x}^{(i)}]$.

Recall that for all $i \in \{1, 2, \ldots, k\}$, $\boldsymbol{e}_i \in \{0, 1\}^k$ is a vector where the i-th entry is 1 and all other entries are 0. Following the approach in Section 3.2, all parties hold a degree-t packed Shamir sharing $[\boldsymbol{e}_i]$ for each $i \in \{1, 2, \ldots, k\}$, and the whole sharing $[\boldsymbol{e}_i]$ is known to all parties. The idea of computing $[\boldsymbol{y}]$ is to first compute $\langle \boldsymbol{y} \rangle := \sum_{i=1}^{k} [\boldsymbol{e}_i] \cdot [\boldsymbol{x}^{(i)}]$ and then use the same technique as MULT to do degree reduction. Unlike MULT where both input sharings are not public, for SELECT, each $[\boldsymbol{e}_i]$ is public and is guaranteed to be consistent. We will show that the inconsistency of the input sharings $[\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(k)}]$ can be transformed to additive errors to the secrets $\boldsymbol{y}$. The description of $\mathcal{F}_{\text{select-mal}}$ appears in Functionality 26. We mark the changes in blue for Functionality $\mathcal{F}_{\text{select-mal}}$ compared with that in the semi-honest setting.

Functionality 26: $\mathcal{F}_{\text{select-mal}}$

1. $\mathcal{F}_{\text{select-mal}}$ receives from honest parties their shares of k degree-t packed Shamir sharings $[\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(k)}]$. $\mathcal{F}_{\text{select-mal}}$ also receives a permutation p from honest parties.

2. For all $i \in \{1, 2, \ldots, k\}$, $\mathcal{F}_{\text{select-mal}}$ reconstructs the whole sharings $[\boldsymbol{x}^{(i)}]_{\mathcal{H}}, [\boldsymbol{x}^{(i)}]$ and computes $\boldsymbol{\Delta}^{(i)} := [\boldsymbol{x}^{(i)}] - [\boldsymbol{x}^{(i)}]_{\mathcal{H}}$. Then $\mathcal{F}_{\text{select-mal}}$ sends the shares of $[\boldsymbol{x}^{(i)}]_{\mathcal{H}}$ of corrupted parties and $\boldsymbol{\Delta}^{(i)}$ to the adversary.

3. $\mathcal{F}_{\text{select-mal}}$ reconstructs $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(k)}$. Then $\mathcal{F}_{\text{select-mal}}$ sets $\boldsymbol{y} = (y_1, y_2, \ldots, y_k)$ such that for all $i \in [k]$, $y_{p(i)} = x^{(i)}_{p(i)}$, where $x^{(i)}_j$ is the j-th value of $\boldsymbol{x}^{(i)}$.

4. $\mathcal{F}_{\text{select-mal}}$ receives from the adversary a vector $\boldsymbol{d} \in \mathbb{F}^k$ and sets $\boldsymbol{y} := \boldsymbol{y} + \boldsymbol{d}$.

5. $\mathcal{F}_{\text{select-mal}}$ receives from the adversary a set of shares $\{s_i\}_{i \in \mathcal{C}}$. $\mathcal{F}_{\text{select-mal}}$ samples a random degree-t packed Shamir sharing $[\boldsymbol{y}]_{\mathcal{H}}$ such that for all $P_i \in \mathcal{C}$, the i-th share of $[\boldsymbol{y}]_{\mathcal{H}}$ is s_i.

6. $\mathcal{F}_{\text{select-mal}}$ receives from the adversary a vector $\boldsymbol{\Delta} \in \mathbb{F}^n$ such that for all $P_i \notin \mathcal{H}_{\mathcal{C}}$, $\Delta_i = 0$. $\mathcal{F}_{\text{select-mal}}$ computes $[\boldsymbol{y}] = [\boldsymbol{y}]_{\mathcal{H}} + \boldsymbol{\Delta}$.

7. $\mathcal{F}_{\text{select-mal}}$ distributes the shares of $[\boldsymbol{y}]$ to honest parties.

Lemma 13. *Protocol* Select *(see Protocol 11) securely computes $\mathcal{F}_{\textit{select-mal}}$ in the $\mathcal{F}_{\textit{rand}}$-hybrid model against a fully malicious adversary who controls $t' = t - k + 1$ parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that $\mathcal{H}_{\mathcal{H}} \subset \mathcal{H}$ is a fixed subset of size $t+1$.

Recall that in Lemma 3, we have shown that, for a pair of two random sharings $([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$, given the shares of $[\boldsymbol{r}], \langle \boldsymbol{r} \rangle$ held by corrupted parties, the shares of $\langle \boldsymbol{r} \rangle$ held by honest parties are uniform.

Simulation for Select. Now we describe the construction of the simulator $\mathcal{S}$.

- In the beginning, $\mathcal{S}$ receives from $\mathcal{F}_{\text{select-mal}}$ the shares of $[\boldsymbol{x}^{(i)}]_{\mathcal{H}}$ held by corrupted parties and $\boldsymbol{\Delta}^{(i)}$ for all $i \in \{1, 2, \ldots, k\}$.

- In Step 2, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ and receives the shares $\{(s^{(0)}_i, s^{(1)}_i)\}_{i \in \mathcal{C}}$ from the adversary.

- In Step 3, $\mathcal{S}$ generates a random element in $\mathbb{F}$ for each honest party as its share of $\langle \boldsymbol{e} \rangle$. Since the shares of $\langle \boldsymbol{r} \rangle$ of honest parties in the real world are uniform, the distribution of the shares of $\langle \boldsymbol{e} \rangle$ of honest parties generated by $\mathcal{S}$ is identical to that in the real world. Using the shares of $[\boldsymbol{x}^{(1)}]_{\mathcal{H}}, [\boldsymbol{x}^{(2)}]_{\mathcal{H}}, \ldots, [\boldsymbol{x}^{(k)}]_{\mathcal{H}}$ and $\langle \boldsymbol{r} \rangle$ held by corrupted parties, $\mathcal{S}$ computes the shares of $\langle \boldsymbol{e} \rangle$ that corrupted parties should hold by

$$\langle \boldsymbol{e} \rangle = \sum_{i=1}^{k} [\boldsymbol{e}_{p(i)}] \cdot [\boldsymbol{x}^{(i)}] + \langle \boldsymbol{r} \rangle.$$

  Let

$$\langle \hat{\boldsymbol{e}} \rangle = \sum_{i=1}^{k} [\boldsymbol{e}_{p(i)}] \cdot [\boldsymbol{x}^{(i)}]_{\mathcal{H}} + \langle \boldsymbol{r} \rangle,$$

which corresponds to the case when parties in $\mathcal{H}_{\mathcal{C}}$ use the shares without additive errors. Note that

$$\langle \widehat{\boldsymbol{e}} \rangle = \sum_{i=1}^{k} [\boldsymbol{e}_{p(i)}] \cdot ([\boldsymbol{x}^{(i)}] - \boldsymbol{\Delta}^{(i)}) + \langle \boldsymbol{r} \rangle = \langle \boldsymbol{e} \rangle - \sum_{i=1}^{k} [\boldsymbol{e}_i] * \boldsymbol{\Delta}^{(i)}.$$

Since for all $i \in \{1, 2, \ldots, k\}$, $\mathcal{S}$ learns $\boldsymbol{\Delta}^{(i)}$ and $[\boldsymbol{e}_i]$ is known to all parties (including $\mathcal{S}$), $\mathcal{S}$ computes the whole sharing $\langle \widehat{\boldsymbol{e}} \rangle$ and reconstructs $\widehat{\boldsymbol{e}}$.

- $\mathcal{S}$ honestly follows Step 4 and Step 5. Let $[\boldsymbol{e}']$ denote the sharing distributed by P_1. At the end of Step 5, $\mathcal{S}$ learns the shares of $[\boldsymbol{e}']$ held by honest parties. $\mathcal{S}$ reconstructs the whole sharings $[\boldsymbol{e}']_{\mathcal{H}}, [\boldsymbol{e}']$ and computes the secrets $\boldsymbol{e}'$ of $[\boldsymbol{e}']_{\mathcal{H}}$. $\mathcal{S}$ sets $\boldsymbol{d} = \boldsymbol{e}' - \widehat{\boldsymbol{e}}$ and $\boldsymbol{\Delta} = [\boldsymbol{e}'] - [\boldsymbol{e}']_{\mathcal{H}}$.
- In Step 6, $\mathcal{S}$ computes the shares of $[\boldsymbol{y}]_{\mathcal{H}}$ held by corrupted parties by using the shares of $[\boldsymbol{e}']_{\mathcal{H}}$ and $[\boldsymbol{r}]$ held by corrupted parties.
- Finally, $\mathcal{S}$ sends to $\mathcal{F}_{\text{select-mal}}$ the vector $\boldsymbol{d}$, the shares of $[\boldsymbol{y}]_{\mathcal{H}}$ held by corrupted parties, and the vector $\boldsymbol{\Delta}$.

Analyze the Security of Select. It is clear that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. It is sufficient to show that the shares of honest parties in both worlds have the same distribution.

- In the real world, let $[\boldsymbol{e}']$ denote the degree-t packed Shamir sharing distributed by P_1. Then, the secrets of the output sharing are $\boldsymbol{e}' - \boldsymbol{r}$. In the ideal world, $\mathcal{F}_{\text{select-mal}}$ first computes the secrets $\boldsymbol{y} = \sum_{i=1}^{k} \boldsymbol{e}_{p(i)} * \boldsymbol{x}^{(i)}$. Note that $\boldsymbol{y}$ are the secrets of $\langle \boldsymbol{y} \rangle := \sum_{i=1}^{k} [\boldsymbol{e}_{p(i)}] \cdot [\boldsymbol{x}^{(i)}]_{\mathcal{H}}$. Recall that $\langle \widehat{\boldsymbol{e}} \rangle := \sum_{i=1}^{k} [\boldsymbol{e}_{p(i)}] \cdot [\boldsymbol{x}^{(i)}]_{\mathcal{H}} + \langle \boldsymbol{r} \rangle$. Therefore, $\boldsymbol{y} = \widehat{\boldsymbol{e}} - \boldsymbol{r}$. The simulator computes $\boldsymbol{d} = \boldsymbol{e}' - \widehat{\boldsymbol{e}}$ as above and sends $\boldsymbol{d}$ to $\mathcal{F}_{\text{select-mal}}$. The final secrets are set to be $\boldsymbol{y} := \boldsymbol{y} + \boldsymbol{d} = \widehat{\boldsymbol{e}} - \boldsymbol{r} + \boldsymbol{e}' - \widehat{\boldsymbol{e}} = \boldsymbol{e}' - \boldsymbol{r}$. Thus, the secrets $\boldsymbol{y}$ in both worlds have the same distribution.
- In the real world, the output sharing $[\boldsymbol{y}]$ is computed by $[\boldsymbol{e}'] - [\boldsymbol{r}]$. Then we have $[\boldsymbol{y}] = [\boldsymbol{e}'] - [\boldsymbol{r}]$ and $[\boldsymbol{y}]_{\mathcal{H}} = [\boldsymbol{e}']_{\mathcal{H}} - [\boldsymbol{r}]$ since $[\boldsymbol{r}]$ is guaranteed to be consistent according to $\mathcal{F}_{\text{rand}}$. Recall that the shares of $[\boldsymbol{y}]$ that corrupted parties should hold are the shares of $[\boldsymbol{y}]_{\mathcal{H}}$ of corrupted parties. In the ideal world, $\mathcal{S}$ computes the shares of $[\boldsymbol{y}]_{\mathcal{H}}$ of corrupted parties using the shares of $[\boldsymbol{e}']_{\mathcal{H}}$ and $[\boldsymbol{r}]$ of corrupted parties as above. Therefore, the shares of $[\boldsymbol{y}]$ that corrupted parties should hold in both worlds are identical.
- In the real world, the additive errors to the shares of $[\boldsymbol{y}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ are $[\boldsymbol{y}] - [\boldsymbol{y}]_{\mathcal{H}}$. Note that $[\boldsymbol{y}] - [\boldsymbol{y}]_{\mathcal{H}} = [\boldsymbol{e}'] - [\boldsymbol{e}']_{\mathcal{H}}$. Therefore, the additive errors $\boldsymbol{\Delta} = [\boldsymbol{e}'] - [\boldsymbol{e}']_{\mathcal{H}}$ computed by $\mathcal{S}$ are identical to the additive errors in the real world.

Note that $[\boldsymbol{y}]$ distributed by $\mathcal{F}_{\text{select-mal}}$ is determined by the secrets $\boldsymbol{y}$, the shares of corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. The distributions of them in both worlds are identical. Therefore, the shares of honest parties in both worlds have the same distribution. □

Security of Rand-Perm. In Rand-Perm, we prepare random sharings used to performing permutations on the secrets of degree-t packed Shamir sharings. Let $p(\cdot)$ be a permutation over $\{1, 2, \ldots, k\}$. Recall that each permutation $p(\cdot)$ maps to a permutation matrix $\boldsymbol{M}_p \in \{0,1\}^{k \times k}$ where $(\boldsymbol{M}_p)_{i,j} = 1$ iff $p(i) = j$. To permute a vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_k)$ to $\tilde{\boldsymbol{x}} = (x_{p(1)}, x_{p(2)}, \ldots, x_{p(k)})$, it is equivalent to computing $\tilde{\boldsymbol{x}} = \boldsymbol{M}_p \cdot \boldsymbol{x}$. Recall that to obtain $[\boldsymbol{M}_p \cdot \boldsymbol{x}]$ from $[\boldsymbol{x}]$, all parties need to prepare a pair of random sharings $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}])$.

Given m permutations $p_1, p_2, \ldots, p_m$, the high-level idea of Rand-Perm is to first use Theorem 2 to obtain m permutations $q_1, q_2, \ldots, q_m$ such that for all $i, j \in \{1, 2, \ldots, k\}$, the number of permutations $p \in \{p_1, p_2, \ldots, p_m\}$ such that $p(i) = j$ is the same as the number of permutations $q \in \{q_1, q_2, \ldots, q_m\}$ such that $q(i) = j$, and the number of different permutations in $q_1, q_2, \ldots, q_m$ is bounded by k^2. Then all parties prepare random sharings for $q_1, q_2, \ldots, q_m$. To obtain random sharings for each permutation p_i, all parties use Select to choose suitable components from random sharings for $q_1, q_2, \ldots, q_m$. We refer the readers to Section 4.2 for more details.

We model the functionality $\mathcal{F}_{\text{rand-perm-mal}}$ in Functionality 27. It allows the adversary to add additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ and additive errors to the secrets of the second sharing of each pair. For the protocol Rand-Perm, we replace the invocations of $\mathcal{F}_{\text{select-semi}}$ by invocations of $\mathcal{F}_{\text{select-mal}}$. See Protocol 28 for more details. We mark the changes in blue for Functionality $\mathcal{F}_{\text{rand-perm-mal}}$ and Protocol Rand-Perm-mal compared with those in the semi-honest setting.

Functionality 27: $\mathcal{F}_{\text{rand-perm-mal}}$

1. $\mathcal{F}_{\text{rand-perm-mal}}$ receives from honest parties m permutations $p_1, p_2, \ldots, p_m$ over $\{1, 2, \ldots, k\}$.
2. For all $i \in [m]$, $\mathcal{F}_{\text{rand-perm-mal}}$ receives from the adversary a vector $\boldsymbol{d}^{(i)} \in \mathbb{F}^k$.
3. For all $i \in [m]$, $\mathcal{F}_{\text{rand-perm-mal}}$ receives from the adversary a set of shares $\{(u_j^{(i)}, v_j^{(i)})\}_{j \in \mathcal{C}}$. $\mathcal{F}_{\text{rand-perm-mal}}$ samples a random vector $\boldsymbol{r}^{(i)} \in \mathbb{F}^k$ and samples two degree-t packed Shamir sharings $([\boldsymbol{r}^{(i)}]_{\mathcal{H}}, [\boldsymbol{M}_{p_i} \cdot \boldsymbol{r}^{(i)} + \boldsymbol{d}^{(i)}]_{\mathcal{H}})$ such that for all $P_j \in \mathcal{C}$, the j-th share of $([\boldsymbol{r}^{(i)}]_{\mathcal{H}}, [\boldsymbol{M}_{p_i} \cdot \boldsymbol{r}^{(i)} + \boldsymbol{d}^{(i)}]_{\mathcal{H}})$ is $(u_j^{(i)}, v_j^{(i)})$.
4. For all $i \in [m]$, $\mathcal{F}_{\text{rand-perm-mal}}$ receives from the adversary two vectors $\boldsymbol{\Delta}^{(i,0)}, \boldsymbol{\Delta}^{(i,1)} \in \mathbb{F}^n$ such that for all $P_j \notin \mathcal{H}_{\mathcal{C}}$, $\Delta_j^{(i,0)} = \Delta_j^{(i,1)} = 0$. $\mathcal{F}_{\text{rand-perm-mal}}$ computes $[\boldsymbol{r}^{(i)}] = [\boldsymbol{r}^{(i)}]_{\mathcal{H}} + \boldsymbol{\Delta}^{(i,0)}$ and $[\boldsymbol{M}_{p_i} \cdot \boldsymbol{r}^{(i)} + \boldsymbol{d}^{(i)}] = [\boldsymbol{M}_{p_i} \cdot \boldsymbol{r}^{(i)} + \boldsymbol{d}^{(i)}]_{\mathcal{H}} + \boldsymbol{\Delta}^{(i,1)}$.
5. For all $i \in [m]$, $\mathcal{F}_{\text{rand-perm-mal}}$ distributes the shares of $([\boldsymbol{r}^{(i)}], [\boldsymbol{M}_{p_i} \cdot \boldsymbol{r}^{(i)} + \boldsymbol{d}^{(i)}])$ to honest parties.

Lemma 14. *Protocol* Rand-Perm-mal *(see Protocol 28) securely computes $\mathcal{F}_{\textit{rand-perm-mal}}$ in the $(\mathcal{F}_{\textit{rand}}, \mathcal{F}_{\textit{select-mal}})$-hybrid model against a fully malicious adversary who controls $t' = t - k + 1$ parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that $\mathcal{H}_{\mathcal{H}} \subset \mathcal{H}$ is a fixed subset of size $t + 1$.

Simulation for Rand-Perm-mal. Now we describe the construction of the simulator $\mathcal{S}$.

- In Step 1 and Step 2, $\mathcal{S}$ honestly follows the protocol and computes the permutations $q_1, q_2, \ldots, q_m$.
- In Step 3, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ and receives the shares of corrupted parties when generating the random sharings $([\boldsymbol{r}^{(i)}], [\boldsymbol{M}_{q_i} \cdot \boldsymbol{r}^{(i)}])$ for each permutation q_i. Since the number of corrupted parties is $t - k + 1$, by the property of the degree-t packed Shamir sharing scheme, the secrets $\boldsymbol{r}^{(i)}$ are uniform and independent of the shares held by corrupted parties.
- In Step 4, $\mathcal{S}$ honestly constructs the lists.
- In Step 5, from $\ell = 1$ to m, $\mathcal{S}$ emulates $\mathcal{F}_{\text{select-mal}}$ as follows:
    1. Preparing $[\boldsymbol{v}^{(\ell)}]$ using $\mathcal{F}_{\text{select-mal}}$: Recall that $\mathcal{S}$ learns the shares of each of $[\boldsymbol{r}^{(\ell_1)}], [\boldsymbol{r}^{(\ell_2)}], \ldots, [\boldsymbol{r}^{(\ell_k)}]$ of corrupted parties when emulating $\mathcal{F}_{\text{rand}}$ in Step 3. According to $\mathcal{F}_{\text{rand}}$, these sharings are guaranteed to be consistent. Therefore in Step 2 of $\mathcal{F}_{\text{select-mal}}$, $\mathcal{S}$ sets $\boldsymbol{\Delta}^{(i)}$ to be the all-0 vector and sends to the adversary the shares of $[\boldsymbol{r}^{(\ell_i)}]$ of corrupted parties and $\boldsymbol{\Delta}^{(i)}$. From Step 4 to Step 6, $\mathcal{S}$ receives from the adversary the additive errors to $\boldsymbol{v}^{(\ell)}$ (denoted by $\boldsymbol{\delta}^{(\ell,0)}$), the shares of $[\boldsymbol{v}^{(\ell)}]$ of corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ (denoted by $\boldsymbol{\Delta}^{(\ell,0)}$).

Protocol 28: Rand-Perm-mal

1. Let $p_1, p_2, \ldots, p_m$ be the permutations over $\{1, 2, \ldots, k\}$ that all parties want to prepare random sharings for.

2. All parties use a deterministic algorithm that all parties agree on to compute m permutations $q_1, q_2, \ldots, q_m$ such that

   - For all $i, j \in \{1, 2, \ldots, k\}$, the number of permutations $p \in \{p_1, p_2, \ldots, p_m\}$ such that $p(i) = j$ is the same as the number of permutations $q \in \{q_1, q_2, \ldots, q_m\}$ such that $q(i) = j$.
   - $q_1, q_2, \ldots, q_m$ contain at most k^2 different permutations.

   The existence of such an algorithm is guaranteed by Theorem 2.

3. Suppose $q'_1, q'_2, \ldots, q'_{k^2}$ denote the different permutations in $q_1, q_2, \ldots, q_m$. For all $i \in \{1, 2, \ldots, k^2\}$, let n'_i denote the number of times that q'_i appears in $q_1, q_2, \ldots, q_m$. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare n'_i pairs of random sharings in the form $([\boldsymbol{r}], [\boldsymbol{M}_{q'_i} \cdot \boldsymbol{r}])$ for all $i \in \{1, 2, \ldots, k^2\}$. Note that we have prepared a pair of random sharings for each permutation q_i for all $i \in [m]$. Let $([\boldsymbol{r}^{(i)}], [\boldsymbol{M}_{q_i} \cdot \boldsymbol{r}^{(i)}])$ denote the random sharings for the permutation q_i.

4. For all $i, j \in \{1, 2, \ldots, k\}$, all parties initiate an empty list $L_{i,j}$. From $\ell = 1$ to m, for all $i, j \in \{1, 2, \ldots, k\}$, if $([\boldsymbol{r}^{(\ell)}], [\boldsymbol{M}_{q_\ell} \cdot \boldsymbol{r}^{(\ell)}])$ contains an (i, j)-component, all parties insert $([\boldsymbol{r}^{(\ell)}], [\boldsymbol{M}_{q_\ell} \cdot \boldsymbol{r}^{(\ell)}])$ into the list $L_{i,j}$.

5. From $\ell = 1$ to m, all parties prepare a pair of random sharings for p_ℓ as follows:

   - From $i = 1$ to k, let $([\boldsymbol{r}^{(\ell_i)}], [\boldsymbol{M}_{q_{\ell_i}} \cdot \boldsymbol{r}^{(\ell_i)}])$ denote the first pair of sharings in the list $L_{i,p_\ell(i)}$, and then remove it from $L_{i,p_\ell(i)}$. Note that $([\boldsymbol{r}^{(\ell_i)}], [\boldsymbol{M}_{q_{\ell_i}} \cdot \boldsymbol{r}^{(\ell_i)}])$ contains an $(i, p_\ell(i))$-component, which is not used when preparing random sharings for $p_1, p_2, \ldots, p_{\ell-1}$.
   - Let I denote the identity permutation over $\{1, 2, \ldots, k\}$.
     - All parties invoke $\mathcal{F}_{\text{select-mal}}$ with
       $$[\boldsymbol{r}^{(\ell_1)}], [\boldsymbol{r}^{(\ell_2)}], \ldots, [\boldsymbol{r}^{(\ell_k)}]$$
       and the permutation I. The output is denoted by $[\boldsymbol{v}^{(\ell)}]$.
     - All parties invoke $\mathcal{F}_{\text{select-mal}}$ with
       $$[\boldsymbol{M}_{q_{\ell_1}} \cdot \boldsymbol{r}^{(\ell_1)}], [\boldsymbol{M}_{q_{\ell_2}} \cdot \boldsymbol{r}^{(\ell_2)}], \ldots, [\boldsymbol{M}_{q_{\ell_k}} \cdot \boldsymbol{r}^{(\ell_k)}]$$
       and the permutation p_ℓ. The output is denoted by $[\tilde{\boldsymbol{v}}^{(\ell)}]$. Note that for all $i \in [k]$, $v_i^{(\ell)} = r_i^{(\ell_i)} = (\boldsymbol{M}_{q_{\ell_i}} \cdot \boldsymbol{r}^{(\ell_i)})_{q_{\ell_i}(i)} = (\boldsymbol{M}_{q_{\ell_i}} \cdot \boldsymbol{r}^{(\ell_i)})_{p_\ell(i)} = \tilde{v}^{(\ell)}_{p_\ell(i)}$.

6. All parties take $([\boldsymbol{v}^{(1)}], [\tilde{\boldsymbol{v}}^{(1)}]), ([\boldsymbol{v}^{(2)}], [\tilde{\boldsymbol{v}}^{(2)}]), \ldots, ([\boldsymbol{v}^{(\ell)}], [\tilde{\boldsymbol{v}}^{(\ell)}])$ as output.

2. Preparing $[\tilde{\boldsymbol{v}}^{(\ell)}]$ using $\mathcal{F}_{\text{select-mal}}$: Recall that $\mathcal{S}$ learns the shares of each of $[\boldsymbol{M}_{q_{\ell_1}} \cdot \boldsymbol{r}^{(\ell_1)}], [\boldsymbol{M}_{q_{\ell_2}} \cdot \boldsymbol{r}^{(\ell_2)}], \ldots, [\boldsymbol{M}_{q_{\ell_k}} \cdot \boldsymbol{r}^{(\ell_k)}]$ of corrupted parties when emulating $\mathcal{F}_{\text{rand}}$ in Step 3. According to $\mathcal{F}_{\text{rand}}$, these sharings are guaranteed to be consistent. Therefore in Step 2 of $\mathcal{F}_{\text{select-mal}}$, $\mathcal{S}$ sets $\boldsymbol{\Delta}^{(i)}$ to be the all-0 vector and sends to the adversary the shares of $[\boldsymbol{M}_{q_{\ell_i}} \cdot \boldsymbol{r}^{(\ell_i)}]$ of corrupted parties and $\boldsymbol{\Delta}^{(i)}$. From Step 4 to Step 6, $\mathcal{S}$ receives from the adversary the additive errors to $\tilde{\boldsymbol{v}}^{(\ell)}$ (denoted

by $\boldsymbol{\delta}^{(\ell,1)}$), the shares of $[\tilde{\boldsymbol{v}}^{(\ell)}]$ of corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ (denoted by $\boldsymbol{\Delta}^{(\ell,1)}$).

$\mathcal{S}$ sets $\boldsymbol{d}^{(\ell)} = \boldsymbol{\delta}^{(\ell,1)} - \boldsymbol{M}_{p_\ell} \cdot \boldsymbol{\delta}^{(\ell,0)}$.

- Finally, $\mathcal{S}$ sends to $\mathcal{F}_{\text{rand-perm-mal}}$ the vector $\boldsymbol{d}^{(\ell)}$, the shares of $[\boldsymbol{v}^{(\ell)}], [\tilde{\boldsymbol{v}}^{(\ell)}]$ of corrupted parties, and the vectors $\boldsymbol{\Delta}^{(\ell,0)}, \boldsymbol{\Delta}^{(\ell,1)}$.

Analyze the Security of Rand-Perm-mal. Note that throughout the protocol, corrupted parties do not receive any messages from honest parties. The only messages the adversary receive are from $\mathcal{F}_{\text{select-mal}}$, which contain the shares of the input sharings of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Since the input sharings of $\mathcal{F}_{\text{select-mal}}$ directly come from $\mathcal{F}_{\text{rand}}$, the sharings are guaranteed to be consistent and the simulator $\mathcal{S}$ learns the shares of corrupted parties when emulating $\mathcal{F}_{\text{rand}}$. Therefore, $\mathcal{S}$ perfectly simulates the behaviors of honest parties and the $\mathcal{F}_{\text{select-mal}}$. It is sufficient to argue that the distributions of the output in both worlds are identical.

- In the real world, the adversary add additive errors $\boldsymbol{\delta}^{(\ell,0)}, \boldsymbol{\delta}^{(\ell,1)}$ to the secrets of $([\boldsymbol{v}^{(\ell)}], [\tilde{\boldsymbol{v}}^{(\ell)}])$. By the correctness of RAND-PERM-MAL, $\boldsymbol{v}^{(\ell)} - \boldsymbol{\delta}^{(\ell,0)}$ is a uniform vector, and $\boldsymbol{M}_{p_\ell} \cdot (\boldsymbol{v}^{(\ell)} - \boldsymbol{\delta}^{(\ell,0)}) = \tilde{\boldsymbol{v}}^{(\ell)} - \boldsymbol{\delta}^{(\ell,1)}$. Since $\boldsymbol{\delta}^{(\ell,0)}$ is independent of $\boldsymbol{v}^{(\ell)} - \boldsymbol{\delta}^{(\ell,0)}$, $\boldsymbol{v}^{(\ell)}$ is also a uniform vector. The additive errors to the second secrets can be computed by
$$\tilde{\boldsymbol{v}}^{(\ell)} - \boldsymbol{M}_{p_\ell} \cdot \boldsymbol{v}^{(\ell)} = \boldsymbol{\delta}^{(\ell,1)} - \boldsymbol{M}_{p_\ell} \cdot \boldsymbol{\delta}^{(\ell,0)}.$$
In the ideal world, $\boldsymbol{v}^{(\ell)}$ is a uniform vector. The simulator computes $\boldsymbol{d}^{(\ell)} = \boldsymbol{\delta}^{(\ell,1)} - \boldsymbol{M}_{p_\ell} \cdot \boldsymbol{\delta}^{(\ell,0)}$ as above and sends to $\mathcal{F}_{\text{rand-perm-mal}}$. Then the second secrets are set to be $\boldsymbol{M}_{p_\ell} \cdot \boldsymbol{v}^{(\ell)} + \boldsymbol{d}^{(\ell)} = \tilde{\boldsymbol{v}}^{(\ell)}$. Therefore the secrets of the output sharings in both worlds have the same distribution.
- In the real world, the shares of $[\boldsymbol{v}^{(\ell)}], [\tilde{\boldsymbol{v}}^{(\ell)}]$ of corrupted parties are chosen by the adversary. In the ideal world, $\mathcal{S}$ learns these shares when emulating $\mathcal{F}_{\text{select-mal}}$ and sends them to $\mathcal{F}_{\text{rand-perm-mal}}$. Therefore, the shares of the output sharings of corrupted parties in both worlds have the same distribution.
- In the real world, the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ are chosen by the adversary. In the ideal world, $\mathcal{S}$ learns these additive errors when emulating $\mathcal{F}_{\text{select-mal}}$ and sends them to $\mathcal{F}_{\text{rand-perm-mal}}$. Therefore, the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ in both worlds have the same distribution.

Thus, the shares of honest parties in both worlds have the same distribution. $\square$

Security of Permute. We model the functionality $\mathcal{F}_{\text{permute-mal}}$ in Functionality 29. In PERMUTE, we replace the invocation of $\mathcal{F}_{\text{rand-perm-semi}}$ by the invocation of $\mathcal{F}_{\text{rand-perm-mal}}$. See Protocol 30 for more details. We mark the changes in blue for Functionality $\mathcal{F}_{\text{permute-mal}}$ and Protocol PERMUTE-MAL compared with those in the semi-honest setting.

Lemma 15. *Protocol* PERMUTE-MAL *(see Protocol 30) securely computes $\mathcal{F}_{permute\text{-}mal}$ in the $(\mathcal{F}_{rand}, \mathcal{F}_{rand\text{-}perm\text{-}mal})$-hybrid model against a fully malicious adversary who controls $t' = t - k + 1$ parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that $\mathcal{H}_{\mathcal{H}} \subset \mathcal{H}$ is a fixed subset of size $t+1$.

Let $[\boldsymbol{r}]$ be a random degree-t packed Shamir sharing, and $\langle \boldsymbol{0} \rangle$ be a random degree-$2t$ packed Shamir sharing of $\boldsymbol{0} \in \mathbb{F}^k$. In Lemma 6, we have shown that, given the shares of $[\boldsymbol{r}]$ and $\langle \boldsymbol{0} \rangle$ held by corrupted parties, the shares of $\langle \boldsymbol{r} \rangle := [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle$ held by honest parties are uniform.

Functionality 29: $\mathcal{F}_{\text{permute-mal}}$

1. $\mathcal{F}_{\text{permute-mal}}$ receives a permutation p and the shares of a degree-t packed Shamir sharing $[\boldsymbol{x}]$ from honest parties.
2. $\mathcal{F}_{\text{permute-mal}}$ reconstructs the whole sharings $[\boldsymbol{x}]_{\mathcal{H}}, [\boldsymbol{x}]$ and computes $\boldsymbol{\Delta}_x := [\boldsymbol{x}] - [\boldsymbol{x}]_{\mathcal{H}}$. Then $\mathcal{F}_{\text{permute-mal}}$ sends the shares of $[\boldsymbol{x}]_{\mathcal{H}}$ of corrupted parties and $\boldsymbol{\Delta}_x$ to the adversary.
3. $\mathcal{F}_{\text{permute-mal}}$ reconstructs the secrets $\boldsymbol{x}$ from the shares of honest parties, and computes $\tilde{\boldsymbol{x}} = \boldsymbol{M}_p \cdot \boldsymbol{x}$.
4. $\mathcal{F}_{\text{permute-mal}}$ receives from the adversary a vector $\boldsymbol{d} \in \mathbb{F}^k$ and sets $\tilde{\boldsymbol{x}} := \tilde{\boldsymbol{x}} + \boldsymbol{d}$.
5. $\mathcal{F}_{\text{permute-mal}}$ receives from the adversary a set of shares $\{s_i\}_{i \in \mathcal{C}}$. $\mathcal{F}_{\text{permute-mal}}$ samples a random degree-t packed Shamir sharing $[\tilde{\boldsymbol{x}}]_{\mathcal{H}}$ such that for all $P_i \in \mathcal{C}$, the i-th share of $[\tilde{\boldsymbol{x}}]_{\mathcal{H}}$ is s_i.
6. $\mathcal{F}_{\text{permute-mal}}$ receives from the adversary a vector $\boldsymbol{\Delta}_{\tilde{x}} \in \mathbb{F}^n$ such that for all $P_i \notin \mathcal{H}_{\mathcal{C}}$, $(\boldsymbol{\Delta}_{\tilde{x}})_i = 0$. $\mathcal{F}_{\text{permute-mal}}$ computes $[\tilde{\boldsymbol{x}}] = [\tilde{\boldsymbol{x}}]_{\mathcal{H}} + \boldsymbol{\Delta}_{\tilde{x}}$.
7. $\mathcal{F}_{\text{permute-mal}}$ distributes the shares of $[\tilde{\boldsymbol{x}}]$ to honest parties.

Protocol 30: Permute-mal

1. Let $[\boldsymbol{x}]$ denote the input degree-t packed Shamir sharing and $p(\cdot)$ denote the permutation all parties want to perform on $\boldsymbol{x}$.
2. All parties invoke $\mathcal{F}_{\text{rand-perm-mal}}$ with p to prepare a pair of random sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare a random degree-$2t$ packed Shamir sharing $\langle \boldsymbol{0} \rangle$.
3. All parties locally compute $\langle \boldsymbol{e} \rangle := [\boldsymbol{x}] + [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle$.
4. All parties send their shares of $\langle \boldsymbol{e} \rangle$ to the first party P_1.
5. P_1 reconstructs the secrets $\boldsymbol{e}$, and computes $\tilde{\boldsymbol{e}} = \boldsymbol{M}_p \cdot \boldsymbol{e}$. Then P_1 generates a random degree-t packed Shamir sharing $[\tilde{\boldsymbol{e}}]$, and distributes the shares to other parties.
6. All parties locally compute $[\tilde{\boldsymbol{x}}] := [\tilde{\boldsymbol{e}}] - [\tilde{\boldsymbol{r}}]$.

Simulation for Permute-mal. Now we describe the construction of $\mathcal{S}$.

- In the beginning, $\mathcal{S}$ receives from $\mathcal{F}_{\text{permute-mal}}$ the shares of $[\boldsymbol{x}]$ of corrupted parties and the additive errors $\boldsymbol{\Delta}_x$ to the shares of parties in $\mathcal{H}_{\mathcal{C}}$.
- In Step 2, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand-perm-mal}}$ and $\mathcal{F}_{\text{rand}}$, and receives the shares of $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ and $\langle \boldsymbol{0} \rangle$ held by corrupted parties. In addition, during the emulation of $\mathcal{F}_{\text{rand-perm-mal}}$, $\mathcal{S}$ receives the additive errors $\boldsymbol{\delta}$ to the secrets $\tilde{\boldsymbol{r}}$ and the additive errors $\boldsymbol{\Delta}^{(0)}, \boldsymbol{\Delta}^{(1)}$ to the shares of $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ of parties in $\mathcal{H}_{\mathcal{C}}$.
- In Step 3, for each honest party, $\mathcal{S}$ generates a random element in $\mathbb{F}$ as its share of $\langle \boldsymbol{e} \rangle$. Note that $\langle \boldsymbol{e} \rangle = [\boldsymbol{x}] + [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle = [\boldsymbol{x}] + \boldsymbol{\Delta}^{(0)} + [\boldsymbol{r}]_{\mathcal{H}} + \langle \boldsymbol{0} \rangle$. As we argued above, the share of $\langle \boldsymbol{r} \rangle := [\boldsymbol{r}]_{\mathcal{H}} + \langle \boldsymbol{0} \rangle$ held by each honest party is uniform. Therefore, the share of $\langle \boldsymbol{e} \rangle$ held by each honest party is also uniform. $\mathcal{S}$ computes the shares of $\langle \boldsymbol{e} \rangle$ that corrupted parties should hold by

$$\langle \boldsymbol{e} \rangle = [\boldsymbol{x}] + [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle.$$

Let

$$\langle \widehat{\boldsymbol{e}} \rangle = [\boldsymbol{x}]_{\mathcal{H}} + [\boldsymbol{r}]_{\mathcal{H}} + \langle \boldsymbol{0} \rangle,$$

which corresponds to the case when parties in $\mathcal{H}_{\mathcal{C}}$ use the shares without additive errors. Note that

$$\langle \widehat{\boldsymbol{e}} \rangle = [\boldsymbol{x}] - \boldsymbol{\Delta}_x + [\boldsymbol{r}] - \boldsymbol{\Delta}^{(0)} + \langle \boldsymbol{0} \rangle = \langle \boldsymbol{e} \rangle - \boldsymbol{\Delta}_x - \boldsymbol{\Delta}^{(0)}.$$

Since $\mathcal{S}$ received $\boldsymbol{\Delta}_x$ from $\mathcal{F}_{\text{permute-mal}}$ in the beginning and received $\boldsymbol{\Delta}^{(0)}$ when emulating $\mathcal{F}_{\text{rand-perm-mal}}$ in Step 2, $\mathcal{S}$ computes the whole sharing $\langle \widehat{\boldsymbol{e}} \rangle$ and reconstructs $\widehat{\boldsymbol{e}}$.

- In Step 4, Step 5 and Step 6, $\mathcal{S}$ honestly follows the protocol. Let $[\tilde{\boldsymbol{e}}']$ denote the sharing distributed by P_1. At the end of Step 5, $\mathcal{S}$ learns the shares of $[\tilde{\boldsymbol{e}}']$ held by honest parties. $\mathcal{S}$ reconstructs the whole sharings $[\tilde{\boldsymbol{e}}']_{\mathcal{H}}, [\tilde{\boldsymbol{e}}']$ and computes the secrets $\tilde{\boldsymbol{e}}'$ of $[\tilde{\boldsymbol{e}}']_{\mathcal{H}}$. $\mathcal{S}$ sets $\boldsymbol{d} := \tilde{\boldsymbol{e}}' - \boldsymbol{M}_p \cdot \widehat{\boldsymbol{e}} - \boldsymbol{\delta}$, where $\boldsymbol{\delta}$ are the additive errors to the secrets $\tilde{\boldsymbol{r}}$ that $\mathcal{S}$ received when emulating $\mathcal{F}_{\text{rand-perm-mal}}$ in Step 2, and $\boldsymbol{\Delta}_{\tilde{x}} := [\tilde{\boldsymbol{e}}'] - [\tilde{\boldsymbol{e}}']_{\mathcal{H}} - \boldsymbol{\Delta}^{(1)}$, where $\boldsymbol{\Delta}^{(1)}$ are the additive errors to the shares of $[\tilde{\boldsymbol{r}}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ that $\mathcal{S}$ received when emulating $\mathcal{F}_{\text{rand-perm-mal}}$ in Step 2.
- In Step 6, $\mathcal{S}$ computes the shares of $[\tilde{\boldsymbol{x}}]$ held by corrupted parties using the shares of $[\tilde{\boldsymbol{e}}']_{\mathcal{H}}$ and $[\tilde{\boldsymbol{r}}]$ held by corrupted parties.
- Finally, $\mathcal{S}$ sends to $\mathcal{F}_{\text{permute-mal}}$ the vector $\boldsymbol{d}$, the shares of $[\tilde{\boldsymbol{x}}]$ held by corrupted parties, and the vector $\boldsymbol{\Delta}_{\tilde{x}}$.

Analyze the Security of Permute-mal. Throughout the protocol, the only place where honest parties need to send messages to corrupted parties is in Step 4 when P_1 is corrupted. As we argued above, in the real world, the share of $\langle \boldsymbol{e} \rangle$ of each honest party is uniformly random. In the ideal world, the simulator $\mathcal{S}$ samples a uniform element as the share of $\langle \boldsymbol{e} \rangle$ of each honest party. Therefore, $\mathcal{S}$ perfectly simulates the behaviors of honest parties. It is sufficient to argue that the distributions of the output in both worlds are identical.

- In the real world, the adversary adds additive errors $\boldsymbol{\delta}$ to the secrets of $[\tilde{\boldsymbol{r}}]$ in Step 2. Therefore, we have $\tilde{\boldsymbol{r}} - \boldsymbol{\delta} = \boldsymbol{M}_p \cdot \boldsymbol{r}$. Let $[\tilde{\boldsymbol{e}}']$ denote the sharing distributed by P_1. Then the secrets of the output sharing are
$$\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{e}}' - \tilde{\boldsymbol{r}} = \tilde{\boldsymbol{e}}' - \boldsymbol{M}_p \cdot \boldsymbol{r} - \boldsymbol{\delta}.$$
In the ideal world, $\mathcal{F}_{\text{permute-mal}}$ first computes $\tilde{\boldsymbol{x}} = \boldsymbol{M}_p \cdot \boldsymbol{x}$. Recall that during the simulation, we set
$$\langle \widehat{\boldsymbol{e}} \rangle = [\boldsymbol{x}]_{\mathcal{H}} + [\boldsymbol{r}]_{\mathcal{H}} + \langle \boldsymbol{0} \rangle.$$
Then $\widehat{\boldsymbol{e}} = \boldsymbol{x} + \boldsymbol{r}$, which means $\boldsymbol{x} = \widehat{\boldsymbol{e}} - \boldsymbol{r}$. The simulator computes $\boldsymbol{d} = \tilde{\boldsymbol{e}}' - \boldsymbol{M}_p \cdot \widehat{\boldsymbol{e}} - \boldsymbol{\delta}$ as above and sends it to $\mathcal{F}_{\text{permute-mal}}$. The final secrets are set to be
$$\tilde{\boldsymbol{x}} := \tilde{\boldsymbol{x}} + \boldsymbol{d} = \boldsymbol{M}_p \cdot (\widehat{\boldsymbol{e}} - \boldsymbol{r}) + \tilde{\boldsymbol{e}}' - \boldsymbol{M}_p \cdot \widehat{\boldsymbol{e}} - \boldsymbol{\delta} = \tilde{\boldsymbol{e}}' - \boldsymbol{M}_p \cdot \boldsymbol{r} - \boldsymbol{\delta}.$$
Thus, the secrets of the output sharing in both worlds are identical.
- In the real world, $[\tilde{\boldsymbol{x}}] = [\tilde{\boldsymbol{e}}'] - [\tilde{\boldsymbol{r}}]$. The shares of $[\tilde{\boldsymbol{x}}]$ of corrupted parties can be computed using the shares of $[\tilde{\boldsymbol{e}}']$ and $[\tilde{\boldsymbol{r}}]$ held by corrupted parties. In the ideal world, $\mathcal{S}$ learns the shares of $[\tilde{\boldsymbol{r}}]$ of corrupted parties when emulating $\mathcal{F}_{\text{rand-perm-mal}}$ in Step 2, and $\mathcal{S}$ computes the shares of $[\tilde{\boldsymbol{e}}']$ of corrupted parties using the shares of honest parties received from P_1 in Step 5. $\mathcal{S}$ computes the shares of $[\tilde{\boldsymbol{x}}]$ of corrupted parties accordingly and sends them to $\mathcal{F}_{\text{permute-mal}}$. Therefore, the shares of the output sharing of corrupted parties in both worlds have the same distributions.
- In the real world, the additive errors to the shares of $[\tilde{\boldsymbol{x}}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ can be computed by
$$[\tilde{\boldsymbol{x}}] - [\tilde{\boldsymbol{x}}]_{\mathcal{H}} = ([\tilde{\boldsymbol{e}}'] - [\tilde{\boldsymbol{e}}']_{\mathcal{H}}) - ([\tilde{\boldsymbol{r}}] - [\tilde{\boldsymbol{r}}]_{\mathcal{H}}) = [\tilde{\boldsymbol{e}}'] - [\tilde{\boldsymbol{e}}']_{\mathcal{H}} - \boldsymbol{\Delta}^{(1)}.$$
They are exactly how $\mathcal{S}$ computes the additive errors $\boldsymbol{\Delta}_{\tilde{x}}$. Therefore, the additive errors to the shares of the output sharing of parties in $\mathcal{H}_{\mathcal{C}}$ in both worlds have the same distribution.

Thus, the shares of honest parties in both worlds have the same distribution. $\square$

6.3 Security of Rand-Pattern and Fan-out

Security of Rand-Pattern. In Rand-Pattern, we prepare random sharings used to evaluate fan-out gates (i.e., copying each wire the number of times it used in later layers). We first recall two definitions in Section 4.4.

- For all $i_1, i_2 \in \{1, 2, \ldots, k\}$, we say a pair of degree-t packed Shamir sharings $([\boldsymbol{x}], [\boldsymbol{y}])$ contains an (i_1, i_2)-block if for all $i_1 \leq j \leq i_2$ the secrets of these two sharings satisfy that $y_j = x_{i_1}$. We say a point $j \in \{1, 2, \ldots, k\}$ is covered by an (i_1, i_2)-block if $i_1 \leq j \leq i_2$.
- A pattern π is defined to be a list of blocks such that for all $j \in \{1, 2, \ldots, k\}$, j is covered by exactly one block in π.

For a pattern π, we say a pair of random degree-t packed Shamir sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ corresponds to π if for all (i_1, i_2)-block in π, $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ contains an (i_1, i_2)-block. Given m patterns $\pi_1, \pi_2, \ldots, \pi_m$, the goal of Rand-Pattern is to prepare m pairs of random sharings which correspond to these patterns.

The high-level idea of Rand-Pattern is to first use Theorem 5 to obtain m patterns $\rho_1, \rho_2, \ldots, \rho_m$ such that for all $i, j \in \{1, 2, \ldots, k\}$, the number of patterns in $\{\pi_1, \pi_2, \ldots, \pi_m\}$ that contain an (i, j)-block is equal to the number of patterns in $\{\rho_1, \rho_2, \ldots, \rho_m\}$ that contain an (i, j)-block, and the number of different patterns in $\rho_1, \rho_2, \ldots, \rho_m$ is bounded by k^2. Then all parties prepare random sharings for $\rho_1, \rho_2, \ldots, \rho_m$. To obtain random sharings for each pattern π_i, all parties use Select to choose suitable blocks from random sharings for $\rho_1, \rho_2, \ldots, \rho_m$. We refer the readers to Section 4.4 for more details.

We model the functionality $\mathcal{F}_{\text{rand-pattern-mal}}$ in Functionality 31. It allows the adversary to add additive errors to the shares of parties in $\mathcal{H}_\mathcal{C}$ and additive errors to the secrets of the second sharing of each pair. For the protocol Rand-Pattern, we replace the invocations of $\mathcal{F}_{\text{select-semi}}$ by invocations of $\mathcal{F}_{\text{select-mal}}$. See Protocol 32 for more details. We mark the changes in blue for Functionality $\mathcal{F}_{\text{rand-pattern-mal}}$ and Protocol Rand-Pattern-mal compared with those in the semi-honest setting.

Functionality 31: $\mathcal{F}_{\text{rand-pattern-mal}}$

1. $\mathcal{F}_{\text{rand-pattern-mal}}$ receives from honest parties m patterns $\pi_1, \pi_2, \ldots, \pi_m$.
2. For all $i \in [m]$, $\mathcal{F}_{\text{rand-pattern-mal}}$ receives from the adversary a vector $\boldsymbol{d}^{(i)} \in \mathbb{F}^k$.
3. For all $i \in [m]$, $\mathcal{F}_{\text{rand-pattern-mal}}$ receives from the adversary a set of shares $\{(u_j^{(i)}, v_j^{(i)})\}_{j \in \mathcal{C}}$. $\mathcal{F}_{\text{rand-pattern-mal}}$ samples a random vector $\boldsymbol{r}^{(i)} \in \mathbb{F}^k$ and computes a vector $\tilde{\boldsymbol{r}}^{(i)}$ such that for all (i_1, i_2)-block in π_i and $i_1 \leq j \leq i_2$, $\tilde{r}_j^{(i)} = r_{i_1}^{(i)}$. Then $\mathcal{F}_{\text{rand-pattern-mal}}$ sets $\tilde{\boldsymbol{r}}^{(i)} := \tilde{\boldsymbol{r}}^{(i)} + \boldsymbol{d}^{(i)}$. $\mathcal{F}_{\text{rand-pattern-mal}}$ samples two degree-t packed Shamir sharings $([\boldsymbol{r}^{(i)}]_\mathcal{H}, [\tilde{\boldsymbol{r}}^{(i)}]_\mathcal{H})$ such that for all $P_j \in \mathcal{C}$, the j-th share of $([\boldsymbol{r}^{(i)}]_\mathcal{H}, [\tilde{\boldsymbol{r}}^{(i)}]_\mathcal{H})$ is $(u_j^{(i)}, v_j^{(i)})$.
4. For all $i \in [m]$, $\mathcal{F}_{\text{rand-pattern-mal}}$ receives from the adversary two vectors $\boldsymbol{\Delta}^{(i,0)}, \boldsymbol{\Delta}^{(i,1)} \in \mathbb{F}^n$ such that for all $P_j \notin \mathcal{H}_\mathcal{C}$, $\Delta_j^{(i,0)} = \Delta_j^{(i,1)} = 0$. $\mathcal{F}_{\text{rand-pattern-mal}}$ computes $[\boldsymbol{r}^{(i)}] = [\boldsymbol{r}^{(i)}]_\mathcal{H} + \boldsymbol{\Delta}^{(i,0)}$ and $[\tilde{\boldsymbol{r}}^{(i)}] = [\tilde{\boldsymbol{r}}^{(i)}]_\mathcal{H} + \boldsymbol{\Delta}^{(i,1)}$.
5. For all $i \in [m]$, $\mathcal{F}_{\text{rand-pattern-mal}}$ distributes the shares of $([\boldsymbol{r}^{(i)}], [\tilde{\boldsymbol{r}}^{(i)}])$ to honest parties.

Lemma 16. *Protocol* Rand-Pattern-mal *(see Protocol 32) securely computes $\mathcal{F}_{\text{rand-pattern-mal}}$ in the $(\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{select-mal}})$-hybrid model against a fully malicious adversary who controls $t' = t - k + 1$ parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that $\mathcal{H}_\mathcal{H} \subset \mathcal{H}$ is a fixed subset of size $t + 1$.

Protocol 32: Rand-Pattern-mal

1. Let $\pi_1, \pi_2, \dots, \pi_m$ be the patterns that all parties want to prepare random sharings for.
2. All parties use a deterministic algorithm that all parties agree on to compute m patterns $\rho_1, \rho_2, \dots, \rho_m$ such that
   - For all $i, j \in \{1, 2, \dots, k\}$, the number of patterns $\pi \in \{\pi_1, \pi_2, \dots, \pi_m\}$ such that (i, j)-block is in π is the same as the number of patterns $\rho \in \{\rho_1, \rho_2, \dots, \rho_m\}$ such that (i, j)-block is in ρ.
   - $\rho_1, \rho_2, \dots, \rho_m$ contain at most k^2 different patterns.

   The existence of such an algorithm is guaranteed by Theorem 5.
3. Suppose $\rho'_1, \rho'_2, \dots, \rho'_{k^2}$ denote the different patterns in $\rho_1, \rho_2, \dots, \rho_m$. For all $i \in \{1, 2, \dots, k^2\}$, let n'_i denote the number of times that ρ'_i appears in $\rho_1, \rho_2, \dots, \rho_m$. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare n'_i pairs of random sharings for the pattern ρ'_i for all $i \in \{1, 2, \dots, k^2\}$. Note that we have prepared a pair of random sharings for pattern ρ_i for all $i \in [m]$. Let $([\boldsymbol{r}^{(i)}], [\tilde{\boldsymbol{r}}^{(i)}])$ denote the random sharings for the pattern ρ_i.
4. For all $i, j \in \{1, 2, \dots, k\}$, all parties initiate an empty list $L_{i,j}$. From $\ell = 1$ to m, for all $i, j \in \{1, 2, \dots, k\}$, if $([\boldsymbol{r}^{(\ell)}], [\tilde{\boldsymbol{r}}^{(\ell)}])$ contains an (i, j)-block, all parties insert $([\boldsymbol{r}^{(\ell)}], [\tilde{\boldsymbol{r}}^{(\ell)}])$ into the list $L_{i,j}$.
5. From $\ell = 1$ to m, all parties prepare a pair of random sharings for π_ℓ as follows:
   - Let s denote the number of blocks in π_ℓ. The blocks in π_ℓ are denoted by $(w_1, w_2 - 1)$-block, $(w_2, w_3 - 1)$-block, $\dots$, $(w_s, w_{s+1} - 1)$-block, where $w_1 = 1$, $w_{s+1} = k + 1$ and $w_1 < w_2 < \dots < w_{s+1}$. From $i = 1$ to s, let $([\boldsymbol{r}^{(\ell_i)}], [\tilde{\boldsymbol{r}}^{(\ell_i)}])$ denote the first pair of sharings in the list $L_{w_i, w_{i+1}-1}$, and then remove it from $L_{w_i, w_{i+1}-1}$. Note that $([\boldsymbol{r}^{(\ell_i)}], [\tilde{\boldsymbol{r}}^{(\ell_i)}])$ contains a $(w_i, w_{i+1} - 1)$-block, which is not used when preparing random sharings for $\pi_1, \pi_2, \dots, \pi_{\ell-1}$.
   - Let I denote the identity permutation over $\{1, 2, \dots, k\}$.
     - All parties initiate an empty list Q. For all $j \in \{1, 2, \dots, k\}$, let i_j denote the index such that $w_{i_j} \le j < w_{i_j+1}$. From $j = 1$ to k, all parties insert $[\boldsymbol{r}^{(\ell_{i_j})}]$ into Q. Then, all parties invoke $\mathcal{F}_{\text{select-mal}}$ with the degree-t packed Shamir sharings in Q and the permutation I. The output is denoted by $[\boldsymbol{v}^{(\ell)}]$.
     - All parties initiate an empty list Q'. From $j = 1$ to k, all parties insert $[\tilde{\boldsymbol{r}}^{(\ell_{i_j})}]$ into Q'. Then, all parties invoke $\mathcal{F}_{\text{select-mal}}$ with the degree-t packed Shamir sharings in Q' and the permutation I. The output is denoted by $[\tilde{\boldsymbol{v}}^{(\ell)}]$. Note that for all $i \in \{1, 2, \dots, s\}$ and $w_i \le j < w_{i+1}$, $v_j^{(\ell)} = r_j^{(\ell_i)}$ and $\tilde{v}_j^{(\ell)} = \tilde{r}_j^{(\ell_i)} = r_{w_i}^{(\ell_i)} = v_{w_i}^{(\ell)}$.
6. All parties take $([\boldsymbol{v}^{(1)}], [\tilde{\boldsymbol{v}}^{(1)}]), ([\boldsymbol{v}^{(2)}], [\tilde{\boldsymbol{v}}^{(2)}]), \dots, ([\boldsymbol{v}^{(\ell)}], [\tilde{\boldsymbol{v}}^{(\ell)}])$ as output.

Simulation for Rand-Pattern-mal. Now we describe the construction of the simulator $\mathcal{S}$.

- In Step 1 and Step 2, $\mathcal{S}$ honestly follows the protocol and computes the patterns $\rho_1, \rho_2, \dots, \rho_m$.
- In Step 3, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ and receives the shares of corrupted parties when generating the random sharings $([\boldsymbol{r}^{(i)}], [\tilde{\boldsymbol{r}}^{(i)}])$ for each pattern ρ_i. Since the number of corrupted parties is $t - k + 1$, by the property of the degree-t packed Shamir sharing scheme, the secrets $\boldsymbol{r}^{(i)}$ are uniform and independent

of the shares held by corrupted parties.

- In Step 4, $\mathcal{S}$ honestly constructs the lists.
- In Step 5, from $\ell = 1$ to m, $\mathcal{S}$ emulates $\mathcal{F}_{\text{select-mal}}$ as follows:
  1. Preparing $[\boldsymbol{v}^{(\ell)}]$ using $\mathcal{F}_{\text{select-mal}}$: Recall that $\mathcal{S}$ learns the shares of each of $[\boldsymbol{r}^{(\ell_{i_1})}], [\boldsymbol{r}^{(\ell_{i_2})}], \ldots, [\boldsymbol{r}^{(\ell_{i_k})}]$ of corrupted parties when emulating $\mathcal{F}_{\text{rand}}$ in Step 3. According to $\mathcal{F}_{\text{rand}}$, these sharings are guaranteed to be consistent. Therefore in Step 2 of $\mathcal{F}_{\text{select-mal}}$, $\mathcal{S}$ sets $\boldsymbol{\Delta}^{(j)}$ to be the all-0 vector and sends to the adversary the shares of $[\boldsymbol{r}^{(\ell_{i_j})}]$ of corrupted parties and $\boldsymbol{\Delta}^{(j)}$. From Step 4 to Step 6, $\mathcal{S}$ receives from the adversary the additive errors to $\boldsymbol{v}^{(\ell)}$ (denoted by $\boldsymbol{\delta}^{(\ell,0)}$), the shares of $[\boldsymbol{v}^{(\ell)}]$ of corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ (denoted by $\boldsymbol{\Delta}^{(\ell,0)}$).
  2. Preparing $[\tilde{\boldsymbol{v}}^{(\ell)}]$ using $\mathcal{F}_{\text{select-mal}}$: Recall that $\mathcal{S}$ learns the shares of each of $[\tilde{\boldsymbol{r}}^{(\ell_{i_1})}], [\tilde{\boldsymbol{r}}^{(\ell_{i_2})}], \ldots, [\tilde{\boldsymbol{r}}^{(\ell_{i_k})}]$ of corrupted parties when emulating $\mathcal{F}_{\text{rand}}$ in Step 3. According to $\mathcal{F}_{\text{rand}}$, these sharings are guaranteed to be consistent. Therefore in Step 2 of $\mathcal{F}_{\text{select-mal}}$, $\mathcal{S}$ sets $\boldsymbol{\Delta}^{(j)}$ to be the all-0 vector and sends to the adversary the shares of $[\tilde{\boldsymbol{r}}^{(\ell_{i_j})}]$ of corrupted parties and $\boldsymbol{\Delta}^{(j)}$. From Step 4 to Step 6, $\mathcal{S}$ receives from the adversary the additive errors to $\tilde{\boldsymbol{v}}^{(\ell)}$ (denoted by $\boldsymbol{\delta}^{(\ell,1)}$), the shares of $[\tilde{\boldsymbol{v}}^{(\ell)}]$ of corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ (denoted by $\boldsymbol{\Delta}^{(\ell,1)}$).

  S computes a vector $\widehat{\boldsymbol{\delta}}^{(\ell,0)}$ such that for all (i_1, i_2)-block in π_ℓ and $i_1 \le j \le i_2$, $\widehat{\delta}_j^{(\ell,0)} = \delta_{i_1}^{(\ell,0)}$. $\mathcal{S}$ sets $\boldsymbol{d}^{(\ell)} = \boldsymbol{\delta}^{(\ell,1)} - \widehat{\boldsymbol{\delta}}^{(\ell,0)}$.
- Finally, $\mathcal{S}$ sends to $\mathcal{F}_{\text{rand-pattern-mal}}$ the vector $\boldsymbol{d}^{(\ell)}$, the shares of $[\boldsymbol{v}^{(\ell)}], [\tilde{\boldsymbol{v}}^{(\ell)}]$ of corrupted parties, and the vectors $\boldsymbol{\Delta}^{(\ell,0)}, \boldsymbol{\Delta}^{(\ell,1)}$.

Analyze the Security of Rand-Pattern-mal. Note that throughout the protocol, corrupted parties do not receive any messages from honest parties. The only messages the adversary receive are from $\mathcal{F}_{\text{select-mal}}$, which contain the shares of the input sharings of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Since the input sharings of $\mathcal{F}_{\text{select-mal}}$ directly come from $\mathcal{F}_{\text{rand}}$, the sharings are guaranteed to be consistent and the simulator $\mathcal{S}$ learns the shares of corrupted parties when emulating $\mathcal{F}_{\text{rand}}$. Therefore, $\mathcal{S}$ perfectly simulates the behaviors of honest parties and the $\mathcal{F}_{\text{select-mal}}$. It is sufficient to argue that the distributions of the output in both worlds are identical.

- In the real world, the adversary add additive errors $\boldsymbol{\delta}^{(\ell,0)}, \boldsymbol{\delta}^{(\ell,1)}$ to the secrets of $([\boldsymbol{v}^{(\ell)}], [\tilde{\boldsymbol{v}}^{(\ell)}])$. By the correctness of Rand-Pattern-mal, $\boldsymbol{v}^{(\ell)} - \boldsymbol{\delta}^{(\ell,0)}$ is a uniform vector, and for all (i_1, i_2)-block in π_ℓ and $i_1 \le j \le i_2$, $(\tilde{\boldsymbol{v}}^{(\ell)} - \boldsymbol{\delta}^{(\ell,1)})_j = (\boldsymbol{v}^{(\ell)} - \boldsymbol{\delta}^{(\ell,0)})_{i_1}$. Since $\boldsymbol{\delta}^{(\ell,0)}$ is independent of $\boldsymbol{v}^{(\ell)} - \boldsymbol{\delta}^{(\ell,0)}$, $\boldsymbol{v}^{(\ell)}$ is also a uniform vector. Let $\widehat{\boldsymbol{v}}^{(\ell)}, \widehat{\boldsymbol{\delta}}^{(\ell,0)}$ be two vectors such that for all (i_1, i_2)-block in π_ℓ and $i_1 \le j \le i_2$, $\widehat{v}_j^{(\ell)} = v_{i_1}^{(\ell)}$ and $\widehat{\delta}_j^{(\ell,0)} = \delta_{i_1}^{(\ell,0)}$. Then $\tilde{\boldsymbol{v}}^{(\ell)} - \boldsymbol{\delta}^{(\ell,1)} = \widehat{\boldsymbol{v}}^{(\ell)} - \widehat{\delta}^{(\ell,0)}$. Note that $\widehat{\boldsymbol{v}}^{(\ell)}$ are the correct secrets of the second sharing corresponds to the secrets $\boldsymbol{v}^{(\ell)}$ of the first sharing. The additive errors to the second secrets can be computed by

  $$\tilde{\boldsymbol{v}}^{(\ell)} - \widehat{\boldsymbol{v}}^{(\ell)} = \boldsymbol{\delta}^{(\ell,1)} - \widehat{\boldsymbol{\delta}}^{(\ell,0)}.$$

  In the ideal world, $\boldsymbol{v}^{(\ell)}$ is a uniform vector. Then $\mathcal{F}_{\text{rand-pattern-mal}}$ computes $\widehat{\boldsymbol{v}}^{(\ell)}$ as above. The simulator computes $\widehat{\boldsymbol{\delta}}^{(\ell,0)}$ and $\boldsymbol{d}^{(\ell)} = \boldsymbol{\delta}^{(\ell,1)} - \widehat{\boldsymbol{\delta}}^{(\ell,0)}$ as above and sends to $\mathcal{F}_{\text{rand-pattern-mal}}$. Then the second secrets are set to be $\widehat{\boldsymbol{v}}^{(\ell)} + \boldsymbol{d}^{(\ell)} = \tilde{\boldsymbol{v}}^{(\ell)}$. Therefore the secrets of the output sharings in both worlds have the same distribution.
- In the real world, the shares of $[\boldsymbol{v}^{(\ell)}], [\tilde{\boldsymbol{v}}^{(\ell)}]$ of corrupted parties are chosen by the adversary. In the ideal world, $\mathcal{S}$ learns these shares when emulating $\mathcal{F}_{\text{select-mal}}$ and sends them to $\mathcal{F}_{\text{rand-pattern-mal}}$. Therefore, the shares of the output sharings of corrupted parties in both worlds have the same distribution.

- In the real world, the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ are chosen by the adversary. In the ideal world, $\mathcal{S}$ learns these additive errors when emulating $\mathcal{F}_{\text{select-mal}}$ and sends them to $\mathcal{F}_{\text{rand-pattern-mal}}$. Therefore, the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ in both worlds have the same distribution.

Thus, the shares of honest parties in both worlds have the same distribution. □

Security of Fan-out. We model the functionality $\mathcal{F}_{\text{fan-out-mal}}$ in Functionality 33. In FAN-OUT, we replace the invocations of $\mathcal{F}_{\text{permute-semi}}$ and $\mathcal{F}_{\text{rand-pattern-semi}}$ by the invocations of $\mathcal{F}_{\text{permute-mal}}$ and $\mathcal{F}_{\text{rand-pattern-mal}}$. See Protocol 34 for more details. We mark the changes in blue for Functionality $\mathcal{F}_{\text{fan-out-mal}}$ and Protocol FAN-OUT-MAL compared with those in the semi-honest setting.

Functionality 33: $\mathcal{F}_{\text{fan-out-mal}}$

1. $\mathcal{F}_{\text{fan-out-mal}}$ receives from honest parties the shares of $[\boldsymbol{x}]$ and a vector $(n_1, n_2, \ldots, n_k)$.
2. $\mathcal{F}_{\text{fan-out-mal}}$ reconstructs the whole sharings $[\boldsymbol{x}]_{\mathcal{H}}, [\boldsymbol{x}]$ and computes $\boldsymbol{\Delta}_x := [\boldsymbol{x}] - [\boldsymbol{x}]_{\mathcal{H}}$. Then $\mathcal{F}_{\text{fan-out-mal}}$ sends the shares of $[\boldsymbol{x}]_{\mathcal{H}}$ of corrupted parties and $\boldsymbol{\Delta}_x$ to the adversary.
3. $\mathcal{F}_{\text{fan-out-mal}}$ reconstructs the secrets $\boldsymbol{x} = (x_1, x_2, \ldots, x_k)$. Then $\mathcal{F}_{\text{fan-out-mal}}$ initiates an empty list L. From $i = 1$ to k, $\mathcal{F}_{\text{fan-out-mal}}$ inserts n_i times of x_i into L.
4. Let $m = \frac{n_1+n_2+\ldots+n_k}{k}$. From $i = 1$ to m,
   (a) $\mathcal{F}_{\text{fan-out-mal}}$ sets $\boldsymbol{x}^{(i)}$ to be the vector of the first k elements in L, and then removes the first k elements in L.
   (b) $\mathcal{F}_{\text{fan-out-mal}}$ receives from the adversary a vector $\boldsymbol{d}^{(i)} \in \mathbb{F}^k$ and sets $\boldsymbol{x}^{(i)} := \boldsymbol{x}^{(i)} + \boldsymbol{d}^{(i)}$.
   (c) $\mathcal{F}_{\text{fan-out-mal}}$ receives from the adversary a set of shares $\{s_j^{(i)}\}_{j\in\mathcal{C}}$. $\mathcal{F}_{\text{fan-out-mal}}$ generates a degree-t packed Shamir sharing $[\boldsymbol{x}^{(i)}]_{\mathcal{H}}$ such that the j-th share of $[\boldsymbol{x}^{(i)}]_{\mathcal{H}}$ is $s_j^{(i)}$.
   (d) $\mathcal{F}_{\text{fan-out-mal}}$ receives from the adversary a vector $\boldsymbol{\Delta}^{(i)} \in \mathbb{F}^n$ such that for all $P_j \notin \mathcal{H}_{\mathcal{C}}$, $\Delta_j^{(i)} = 0$. $\mathcal{F}_{\text{permute-mal}}$ computes $[\boldsymbol{x}^{(i)}] = [\boldsymbol{x}^{(i)}]_{\mathcal{H}} + \boldsymbol{\Delta}^{(i)}$.
   (e) $\mathcal{F}_{\text{fan-out-mal}}$ distributes the shares of $[\boldsymbol{x}^{(i)}]$ to honest parties.

Lemma 17. *Protocol* FAN-OUT-MAL *(see Protocol 34) securely computes* $\mathcal{F}_{\textit{fan-out-mal}}$ *in the* $(\mathcal{F}_{\textit{rand}}, \mathcal{F}_{\textit{permute-mal}}, \mathcal{F}_{\textit{rand-pattern-mal}})$*-hybrid model against a fully malicious adversary who controls* $t' = t - k + 1$ *parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that $\mathcal{H}_{\mathcal{H}} \subset \mathcal{H}$ is a fixed subset of size $t+1$.

Let $[\boldsymbol{r}]$ be a random degree-t packed Shamir sharing, and $\langle \boldsymbol{0} \rangle$ be a random degree-$2t$ packed Shamir sharing of $\boldsymbol{0} \in \mathbb{F}^k$. In Lemma 6, we have shown that, given the shares of $[\boldsymbol{r}]$ and $\langle \boldsymbol{0} \rangle$ held by corrupted parties, the shares of $\langle \boldsymbol{r} \rangle := [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle$ held by honest parties are uniform.

Simulation for Fan-out-mal. Now we describe the construction of $\mathcal{S}$.

- In the beginning, $\mathcal{S}$ receives from $\mathcal{F}_{\text{fan-out-mal}}$ the shares of $[\boldsymbol{x}]$ of corrupted parties and the additive errors $\boldsymbol{\Delta}_x$ to the shares of parties in $\mathcal{H}_{\mathcal{C}}$.
- In Step 2, $\mathcal{S}$ determines the secrets of each output degree-t packed Shamir sharing by honestly following the protocol.

Protocol 34: Fan-out-mal

1. Let $[\boldsymbol{x}]$ denote the input degree-t packed Shamir sharing and $(n_1, n_2, \ldots, n_k)$ denote the vector in $\mathbb{N}^k$ where n_i is the number of times of the i-th entry of $\boldsymbol{x}$ that all parties want to copy.

2. All parties run the following steps to determine the secrets of each output sharing:

   (a) Initially, L is set to be an empty list. From $i = 1$ to k, insert n_i times of x_i into L.

   (b) Let $m = \frac{n_1+n_2+\ldots+n_k}{k}$. From $i = 1$ to m, let $\boldsymbol{x}^{(i)}$ be the vector of the first k elements in L, and then removes the first k elements in L. Then, $\boldsymbol{x}^{(i)}$ are the secrets of the i-th output degree-t packed Shamir sharing.

3. For each of $\boldsymbol{y} \in \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$, all parties run the following steps to generate a degree-t packed Shamir sharing of $\boldsymbol{y}$.

   (a) Let s denote the number of different values in $\boldsymbol{y}$. These different values are denoted by $v_1, v_2, \ldots, v_s$. For all $i \in \{1, 2, \ldots, s\}$, let w_i denote the index of the first value of $\boldsymbol{y}$ that is equal to v_i. Let $p(\cdot)$ be a permutation over $\{1, 2, \ldots, k\}$ that all parties agree on such that, after computing $\boldsymbol{x}' = (x_{p(1)}, x_{p(2)}, \ldots, x_{p(k)})$, $x'_{w_i} = v_i$ for all $i \in \{1, 2, \ldots, s\}$.

   (b) All parties invoke $\mathcal{F}_{\text{permute-mal}}$ with input sharing $[\boldsymbol{x}]$ and the permutation p. Let $[\boldsymbol{x}']$ denote the output.

   (c) Let π be a pattern which contains $(w_1, w_2 - 1)$-block, $(w_2, w_3 - 1)$-block, $\ldots$, $(w_s, w_{s+1} - 1)$-block, where $w_{s+1} = k + 1$. All parties invoke $\mathcal{F}_{\text{rand-pattern-mal}}$ with π to prepare a pair of random sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ such that $\boldsymbol{r}$ is uniform in $\mathbb{F}^k$ and for all $(w_i, w_{i+1} - 1)$-block in π and $w_i \le j < w_{i+1}$, $\tilde{r}_j = r_{w_i}$.

   (d) All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare a random degree-$2t$ packed Shamir sharing $\langle \boldsymbol{0} \rangle$.

   (e) All parties locally compute $\langle \boldsymbol{e} \rangle := [\boldsymbol{x}'] + [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle$.

   (f) All parties send their shares of $\langle \boldsymbol{e} \rangle$ to the first party P_1.

   (g) P_1 reconstructs the secrets $\boldsymbol{e}$, and computes $\tilde{\boldsymbol{e}}$ such that for all $(w_i, w_{i+1} - 1)$-block in π and $w_i \le j < w_{i+1}$, $\tilde{e}_j = e_{w_i}$. Then P_1 generates a random degree-t packed Shamir sharing $[\tilde{\boldsymbol{e}}]$, and distributes the shares to other parties.

   (h) All parties locally compute $[\boldsymbol{y}] := [\tilde{\boldsymbol{e}}] - [\tilde{\boldsymbol{r}}]$.

- In Step 3.(b), let p denote the permutation all parties want to perform on $\boldsymbol{x}$. $\mathcal{S}$ emulates $\mathcal{F}_{\text{permute-mal}}$ as follows: $\mathcal{S}$ first sends to the adversary the shares of $[\boldsymbol{x}]$ of corrupted parties and the additive errors $\boldsymbol{\Delta}_x$ to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Note that they are learnt in the beginning from $\mathcal{F}_{\text{fan-out-mal}}$. Let $[\boldsymbol{x}']$ denote the output sharing. $\mathcal{S}$ receives from the adversary the additive errors $\boldsymbol{\delta}_{x'}$ to the secrets $\boldsymbol{x}'$, the shares of $[\boldsymbol{x}']$ held by corrupted parties, and the additive errors $\boldsymbol{\Delta}_{x'}$ to the shares of $[\boldsymbol{x}']$ of parties in $\mathcal{H}_{\mathcal{C}}$. Then the correct secrets of the output sharing should be $\boldsymbol{x}' - \boldsymbol{\delta}_{x'}$.

- In Step 3.(c), let π denote the pattern all parties want to generate random sharings for. $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand-pattern-mal}}$ and receives from the adversary the additive errors $\boldsymbol{\delta}_{\tilde{r}}$ to the secrets $\tilde{\boldsymbol{r}}$, the shares of $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ held by corrupted parties, and the additive errors $\boldsymbol{\Delta}^{(0)}, \boldsymbol{\Delta}^{(1)}$ to the shares of $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ of parties in $\mathcal{H}_{\mathcal{C}}$.

- In Step 3.(d), $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ and receives the shares of $\langle \boldsymbol{0} \rangle$ of corrupted parties.

- In Step 3.(e), for each honest party, $\mathcal{S}$ generates a random element in $\mathbb{F}$ as its share of $\langle \boldsymbol{e} \rangle$. Note that

$\langle \boldsymbol{e} \rangle = [\boldsymbol{x}'] + [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle = [\boldsymbol{x}'] + \boldsymbol{\Delta}^{(0)} + [\boldsymbol{r}]_{\mathcal{H}} + \langle \boldsymbol{0} \rangle$. As we argued above, the share of $\langle \boldsymbol{r} \rangle := [\boldsymbol{r}]_{\mathcal{H}} + \langle \boldsymbol{0} \rangle$ held by each honest party is uniform. Therefore, the share of $\langle \boldsymbol{e} \rangle$ held by each honest party is also uniform. $\mathcal{S}$ computes the shares of $\langle \boldsymbol{e} \rangle$ that corrupted parties should hold by

$$\langle \boldsymbol{e} \rangle = [\boldsymbol{x}'] + [\boldsymbol{r}] + \langle \boldsymbol{0} \rangle.$$

Let

$$\langle \widehat{\boldsymbol{e}} \rangle = [\boldsymbol{x}']_{\mathcal{H}} + [\boldsymbol{r}]_{\mathcal{H}} + \langle \boldsymbol{0} \rangle,$$

which corresponds to the case when parties in $\mathcal{H}_{\mathcal{C}}$ use the shares without additive errors. Note that

$$\langle \widehat{\boldsymbol{e}} \rangle = [\boldsymbol{x}'] - \boldsymbol{\Delta}_{x'} + [\boldsymbol{r}] - \boldsymbol{\Delta}^{(0)} + \langle \boldsymbol{0} \rangle = \langle \boldsymbol{e} \rangle - \boldsymbol{\Delta}_{x'} - \boldsymbol{\Delta}^{(0)}.$$

Since $\mathcal{S}$ received $\boldsymbol{\Delta}_{x'}$ from $\mathcal{F}_{\text{fan-out-mal}}$ in the beginning and received $\boldsymbol{\Delta}^{(0)}$ when emulating $\mathcal{F}_{\text{rand-pattern-mal}}$ in Step 3.(c), $\mathcal{S}$ computes the whole sharing $\langle \widehat{\boldsymbol{e}} \rangle$ and reconstructs $\widehat{\boldsymbol{e}}$.

- In Step 3.(f), Step 3.(g) and Step 3.(h), $\mathcal{S}$ honestly follows the protocol. Let $[\tilde{\boldsymbol{e}}']$ denote the sharing distributed by P_1. At the end of Step 5, $\mathcal{S}$ learns the shares of $[\tilde{\boldsymbol{e}}']$ held by honest parties. $\mathcal{S}$ reconstructs the whole sharings $[\tilde{\boldsymbol{e}}']_{\mathcal{H}}, [\tilde{\boldsymbol{e}}']$ and computes the secrets $\tilde{\boldsymbol{e}}'$ of $[\tilde{\boldsymbol{e}}']_{\mathcal{H}}$. $\mathcal{S}$ computes two vectors $\widehat{\boldsymbol{e}}'$ and $\tilde{\boldsymbol{\delta}}_{x'}$ such that for all (i_1, i_2)-block in π and $i_1 \leq j \leq i_2$, $\widehat{e}'_j = \widehat{e}_{i_1}$ and $(\tilde{\boldsymbol{\delta}}_{x'})_j = (\boldsymbol{\delta}_{x'})_{i_1}$. $\mathcal{S}$ sets $\boldsymbol{d} := \tilde{\boldsymbol{e}}' - \widehat{\boldsymbol{e}}' + \tilde{\boldsymbol{\delta}}_{x'} - \boldsymbol{\delta}_{\tilde{r}}$, where $\boldsymbol{\delta}_{\tilde{r}}$ are the additive errors to the secrets $\tilde{\boldsymbol{r}}$ that $\mathcal{S}$ received when emulating $\mathcal{F}_{\text{rand-pattern-mal}}$ in Step 3.(c), and $\boldsymbol{\Delta} := [\tilde{\boldsymbol{e}}'] - [\tilde{\boldsymbol{e}}']_{\mathcal{H}} - \boldsymbol{\Delta}^{(1)}$, where $\boldsymbol{\Delta}^{(1)}$ are the additive errors to the shares of $[\tilde{\boldsymbol{r}}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ that $\mathcal{S}$ received when emulating $\mathcal{F}_{\text{rand-pattern-mal}}$ in Step 3.(c).
- In Step 3.(h), $\mathcal{S}$ computes the shares of $[\boldsymbol{y}]$ held by corrupted parties using the shares of $[\tilde{\boldsymbol{e}}']_{\mathcal{H}}$ and $[\tilde{\boldsymbol{r}}]$ held by corrupted parties.
- Finally, $\mathcal{S}$ sends to $\mathcal{F}_{\text{fan-out-mal}}$ the vector $\boldsymbol{d}$, the shares of $[\boldsymbol{y}]$ held by corrupted parties, and the vector $\boldsymbol{\Delta}$.

Analyze the Security of Fan-out-mal. Throughout the protocol, there are two places where honest parties or the functionality need to send messages to corrupted parties or the adversary. In Step 3.(b), the functionality $\mathcal{F}_{\text{permute-mal}}$ needs to send to the adversary the shares of $[\boldsymbol{x}]$ of corrupted parties and the additive errors of the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Note that $\mathcal{S}$ received them from $\mathcal{F}_{\text{fan-out-mal}}$ in the beginning. In Step 3.(f), when P_1 is corrupted, honest parties need to send their shares of $\langle \boldsymbol{e} \rangle$ to the corrupted P_1. As we argued above, in the real world, the share of $\langle \boldsymbol{e} \rangle$ of each honest party is uniformly random. In the ideal world, the simulator $\mathcal{S}$ samples a uniform element as the share of $\langle \boldsymbol{e} \rangle$ of each honest party. Therefore, $\mathcal{S}$ perfectly simulates the behaviors of honest parties. It is sufficient to argue that the distributions of the output in both worlds are identical.

- In the real world, the adversary adds additive errors $\boldsymbol{\delta}_{x'}$ to the secrets of $[\boldsymbol{x}']$ in Step 3.(b), and additive errors $\boldsymbol{\delta}_{\tilde{r}}$ to the secrets of $[\tilde{\boldsymbol{r}}]$ in Step 3.(c). Therefore, the correct secrets of $[\boldsymbol{x}']$ and $[\tilde{\boldsymbol{r}}]$ should be $\boldsymbol{x}' - \boldsymbol{\delta}_{x'}$ and $\tilde{\boldsymbol{r}} - \boldsymbol{\delta}_{\tilde{r}}$. In particular, the secrets of $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ satisfy that for all (i_1, i_2)-block in π and $i_1 \leq j \leq i_2$, $(\tilde{\boldsymbol{r}} - \boldsymbol{\delta}_{\tilde{r}})_j = r_{i_1}$. Let $[\tilde{\boldsymbol{e}}']$ denote the sharing distributed by P_1. Then the secrets of the output sharing are
$$\boldsymbol{y} = \tilde{\boldsymbol{e}}' - \tilde{\boldsymbol{r}}.$$
In the ideal world, $\mathcal{F}_{\text{fan-out-mal}}$ first computes $\boldsymbol{y}$ such that for all (i_1, i_2)-block in π and $i_1 \leq j \leq i_2$, $y_j = (\boldsymbol{x}' - \boldsymbol{\delta}_{x'})_{i_1}$. Recall that during the simulation, we set
$$\langle \widehat{\boldsymbol{e}} \rangle = [\boldsymbol{x}']_{\mathcal{H}} + [\boldsymbol{r}]_{\mathcal{H}} + \langle \boldsymbol{0} \rangle.$$
Then $\widehat{\boldsymbol{e}} = \boldsymbol{x}' + \boldsymbol{r}$, which means $\boldsymbol{x}' = \widehat{\boldsymbol{e}} - \boldsymbol{r}$. Recall that $\mathcal{S}$ also computes two vectors $\widehat{\boldsymbol{e}}', \tilde{\boldsymbol{\delta}}_{x'}$ such that for all (i_1, i_2)-block in π and $i_1 \leq j \leq i_2$, $\widehat{e}'_j = \widehat{e}_{i_1}$ and $(\tilde{\boldsymbol{\delta}}_{x'})_j = (\boldsymbol{\delta}_{x'})_{i_1}$. Therefore, $\boldsymbol{y}$ computed by

$\mathcal{F}_{\text{fan-out-mal}}$ are equal to $\widehat{\boldsymbol{e}}' - (\tilde{\boldsymbol{r}} - \boldsymbol{\delta}_{\tilde{r}}) - \tilde{\boldsymbol{\delta}}_{x'}$. The simulator computes $\boldsymbol{d} := \tilde{\boldsymbol{e}}' - \widehat{\boldsymbol{e}}' + \tilde{\boldsymbol{\delta}}_{x'} - \boldsymbol{\delta}_{\tilde{r}}$ as above and sends it to $\mathcal{F}_{\text{fan-out-mal}}$. The final secrets are set to be

$$\boldsymbol{y} := \boldsymbol{y} + \boldsymbol{d} = \widehat{\boldsymbol{e}}' - (\tilde{\boldsymbol{r}} - \boldsymbol{\delta}_{\tilde{r}}) - \tilde{\boldsymbol{\delta}}_{x'} + \tilde{\boldsymbol{e}}' - \widehat{\boldsymbol{e}}' + \tilde{\boldsymbol{\delta}}_{x'} - \boldsymbol{\delta}_{\tilde{r}} = \tilde{\boldsymbol{e}}' - \tilde{\boldsymbol{r}}.$$

Thus, the secrets of the output sharing in both worlds are identical.

- In the real world, $[\boldsymbol{y}] = [\tilde{\boldsymbol{e}}'] - [\tilde{\boldsymbol{r}}]$. The shares of $[\boldsymbol{y}]$ of corrupted parties can be computed using the shares of $[\tilde{\boldsymbol{e}}']$ and $[\tilde{\boldsymbol{r}}]$ held by corrupted parties. In the ideal world, $\mathcal{S}$ learns the shares of $[\tilde{\boldsymbol{r}}]$ of corrupted parties when emulating $\mathcal{F}_{\text{rand-pattern-mal}}$ in Step 3.(c), and $\mathcal{S}$ computes the shares of $[\tilde{\boldsymbol{e}}']$ of corrupted parties using the shares of honest parties received from P_1 in Step 3.(g). $\mathcal{S}$ computes the shares of $[\boldsymbol{y}]$ of corrupted parties accordingly and sends them to $\mathcal{F}_{\text{fan-out-mal}}$. Therefore, the shares of the output sharing of corrupted parties in both worlds have the same distributions.

- In the real world, the additive errors to the shares of $[\boldsymbol{y}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ can be computed by

$$[\boldsymbol{y}] - [\boldsymbol{y}]_{\mathcal{H}} = ([\tilde{\boldsymbol{e}}'] - [\tilde{\boldsymbol{e}}']_{\mathcal{H}}) - ([\tilde{\boldsymbol{r}}] - [\tilde{\boldsymbol{r}}]_{\mathcal{H}}) = [\tilde{\boldsymbol{e}}'] - [\tilde{\boldsymbol{e}}']_{\mathcal{H}} - \boldsymbol{\Delta}^{(1)}.$$

  They are exactly how $\mathcal{S}$ computes the additive errors $\boldsymbol{\Delta}$. Therefore, the additive errors to the shares of the output sharing of parties in $\mathcal{H}_{\mathcal{C}}$ in both worlds have the same distribution.

Thus, the shares of honest parties in both worlds have the same distribution. □

7 Verification of the Computation

In Section 6, we discussed the security of our semi-honest protocols presented in Section 4 against a fully malicious adversary. In particular, we have shown that most of our semi-honest protocols provide perfect privacy against a fully malicious adversary, namely the executions of those protocols do not leak any information to the adversary. To achieve malicious security, our idea is to first run the semi-honest protocol until the output phase, then check the correctness of the computation, and finally reconstruct the output. The perfect privacy of our semi-honest protocols allows us to postpone the verification till the end so that the verification can be done in a batch way.

As for the verification, instead of checking the correctness of every single invocation of every protocol, we view the computation as a composition of two parts: (1) evaluation of the basic gates, i.e., addition gates and multiplication gates, and (2) network routing, i.e., computing input sharings of each layer using the output sharings of previous layers. For the first part, since addition gates are computed without interaction, it is sufficient to only check the correctness of multiplications. We will discuss the verification of multiplications in Section 7.1. For the second part, it includes evaluating the fan-out gates we insert (see Section 4.4), performing permutations on the secrets of sharings (see Section 4.2), and selecting secrets from the output sharings in previous layers (see Section 4.2). We will discuss the verification of the network routing in Section 7.2.

Recall that we are in the client-server model where there are c clients and $n = 2t+1$ parties (servers). Recall that $1 \le k \le t$ is an integer. An adversary is allowed to corrupt $t' = t - k + 1$ parties. We will use the degree-t packed Shamir sharing scheme, which can store k secrets within one sharing. Recall that $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties.

Recall that $\mathcal{H}_{\mathcal{H}} \subset \mathcal{H}$ be a fixed subset of size $t+1$, and $\mathcal{H}_{\mathcal{C}} = \mathcal{H} \backslash \mathcal{H}_{\mathcal{H}}$. For a degree-$t$ packed Shamir sharing held $[\boldsymbol{x}]$ by all parties, we use $[\boldsymbol{x}]_{\mathcal{H}}$ to denote the degree-t packed Shamir sharing determined by the shares of parties in $\mathcal{H}_{\mathcal{H}}$. We assume the shares of $[\boldsymbol{x}]$ of corrupted parties are the same as the shares of $[\boldsymbol{x}]_{\mathcal{H}}$ of corrupted parties. See discussions in Section 6. We use $\boldsymbol{\Delta} = [\boldsymbol{x}] - [\boldsymbol{x}]_{\mathcal{H}}$ to denote the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$.

7.1 Multiplication Verification

We first discuss how to check the correctness of multiplications. At a high-level, we follow the technique in [BBCG+19, GSZ20] to achieve sub-linear communication complexity. The original technique focuses on the case where the number of corrupted parties is $t = (n-1)/2$, and each secret sharing only stores one secret. We combine this technique and the packed secret-sharing technique to verify the multiplications.

We first introduce the functionality we want to achieve. Ideally, the functionality takes m multiplication tuples (where each tuple containing three degree-t packed Shamir sharings corresponding to the input sharings and the output sharing) and checks the correctness of multiplications. To allow an ideal adversary (or the simulator) to generate the views of corrupted parties in the real world, the functionality further provides

1. the additive errors to the shares of parties in $\mathcal{H}_\mathcal{C}$ of each sharing,
2. the additive errors to the multiplication results of each multiplication tuple.

Recall that in $\mathcal{F}_{\text{mult-mal}}$, the adversary is allowed to add additive errors to the shares of parties in $\mathcal{H}_\mathcal{C}$ of the output sharing and additive errors to the secrets of the output sharing. Therefore, additive errors to the shares of parties in $\mathcal{H}_\mathcal{C}$ for each sharing can be provided for free. However, recall that $\mathcal{F}_{\text{mult-mal}}$ computes the multiplication results using the input sharings which contain the additive errors to the shares of parties in $\mathcal{H}_\mathcal{C}$ *instead of using the secrets of the input sharings*. The multiplication results computed by $\mathcal{F}_{\text{mult-mal}}$ are not guaranteed to be correct if the additive errors to the shares of parties in $\mathcal{H}_\mathcal{C}$ of the input sharings are non-zero. It means that the additive errors to the secrets of the output sharing are not equal to the additive errors to the multiplication results. Therefore, in the functionality, the additive errors to the multiplication results for each multiplication tuple are only provided if the additive errors to the shares of parties in $\mathcal{H}_\mathcal{C}$ of the input sharings of each multiplication tuple are 0 (which guarantee that $\mathcal{F}_{\text{mult-mal}}$ computes the correct multiplication results). The functionality $\mathcal{F}_{\text{mult-veri}}$ can be found in Functionality 35.

Functionality 35: $\mathcal{F}_{\text{mult-veri}}$

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

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

2. For all $i \in [m]$, $\mathcal{F}_{\text{mult-veri}}$ receives from honest parties their shares of $[\boldsymbol{x}^{(i)}],[\boldsymbol{y}^{(i)}],[\boldsymbol{z}^{(i)}]$. Then $\mathcal{F}_{\text{mult-veri}}$ reconstructs the whole sharings $[\boldsymbol{x}^{(i)}]_\mathcal{H},[\boldsymbol{x}^{(i)}],[\boldsymbol{y}^{(i)}]_\mathcal{H},[\boldsymbol{y}^{(i)}],[\boldsymbol{z}^{(i)}]_\mathcal{H},[\boldsymbol{z}^{(i)}]$ and computes $\boldsymbol{\Delta}_x^{(i)} := [\boldsymbol{x}^{(i)}]-[\boldsymbol{x}^{(i)}]_\mathcal{H}$, $\boldsymbol{\Delta}_y^{(i)} := [\boldsymbol{y}^{(i)}]-[\boldsymbol{y}^{(i)}]_\mathcal{H}$, and $\boldsymbol{\Delta}_z^{(i)} := [\boldsymbol{z}^{(i)}]-[\boldsymbol{z}^{(i)}]_\mathcal{H}$. $\mathcal{F}_{\text{mult-veri}}$ reconstructs the secrets $\boldsymbol{x}^{(i)},\boldsymbol{y}^{(i)},\boldsymbol{z}^{(i)}$ and computes $\boldsymbol{d}^{(i)} := \boldsymbol{z}^{(i)} - \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$.

3. For all $i \in [m]$, $\mathcal{F}_{\text{mult-veri}}$ sends to the adversary the shares of $[\boldsymbol{x}^{(i)}],[\boldsymbol{y}^{(i)}],[\boldsymbol{z}^{(i)}]$ of corrupted parties and the vectors $\boldsymbol{\Delta}_x^{(i)},\boldsymbol{\Delta}_y^{(i)},\boldsymbol{\Delta}_z^{(i)}$. If $\boldsymbol{\Delta}_x^{(i)} = \boldsymbol{\Delta}_y^{(i)} = \boldsymbol{0}$, $\mathcal{F}_{\text{mult-veri}}$ sends to the adversary the vector $\boldsymbol{d}^{(i)}$.

4. Let $b \in \{\texttt{abort}, \texttt{accept}\}$ denote whether there exists $i \in [m]$ such that at least one of $\boldsymbol{\Delta}_x^{(i)},\boldsymbol{\Delta}_y^{(i)},\boldsymbol{\Delta}_z^{(i)},\boldsymbol{d}^{(i)}$ is non-zero. $\mathcal{F}_{\text{mult-veri}}$ sends b to the adversary and waits for its response.
   - If the adversary replies `continue`, $\mathcal{F}_{\text{mult-veri}}$ sends b to honest parties.
   - If the adversary replies `abort`, $\mathcal{F}_{\text{mult-veri}}$ sends `abort` to honest parties.

7.1.1 Batch-wise Multiplication Verification from [BSFO12].

We first introduce a technique from [BSFO12] which allows all parties to compress the check of m multiplication tuples into the check of a single multiplication tuple with the cost of $O(m)$ additional invocations of $\mathcal{F}_{\text{mult-mal}}$. The original work [BSFO12] focuses on the case where the number of corrupted parties is $t = (n-1)/2$, and each secret sharing only stores one secret. We show that it can be generalized and combined with the packed secret-sharing technique.

Suppose the input multiplication tuples are denoted by

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

At a high-level, all parties will construct three vectors of polynomials $\boldsymbol{F}(\cdot), \boldsymbol{G}(\cdot), \boldsymbol{H}(\cdot)$, where the coefficients of these three polynomials are vectors in $\mathbb{F}^k$, $\boldsymbol{F}, \boldsymbol{G}$ are of degree-$(m-1)$, and $\boldsymbol{H}$ is of degree-$2(m-1)$. The first two polynomials $\boldsymbol{F}, \boldsymbol{G}$ are determined by the following m evaluation points: $\forall i \in [m]$, $\boldsymbol{F}(i) = \boldsymbol{x}^{(i)}$ and $\boldsymbol{G}(i) = \boldsymbol{y}^{(i)}$. For $\boldsymbol{H}$, we want to achieve that

- If the input multiplication tuples are correct and all parties honestly follow the protocol, then $\boldsymbol{H} = \boldsymbol{F} * \boldsymbol{G}$.
- Otherwise, $\boldsymbol{H} \neq \boldsymbol{F} * \boldsymbol{G}$.

In this way, it is sufficient to test whether $\boldsymbol{H} = \boldsymbol{F} * \boldsymbol{G}$, which can be done by testing a random evaluation point. To this end, for all $i \in [m]$, $\boldsymbol{H}(i) = \boldsymbol{z}^{(i)}$. Note that we need additional $m-1$ points to determine the polynomial $\boldsymbol{H}$. For all $i \in \{m+1, \ldots, 2m-1\}$, all parties locally compute $[\boldsymbol{F}(i)], [\boldsymbol{G}(i)]$, which are linear combinations of $[\boldsymbol{x}^{(1)}], \ldots, [\boldsymbol{x}^{(m)}]$ and $[\boldsymbol{y}^{(1)}], \ldots, [\boldsymbol{y}^{(m)}]$ respectively. Then, all parties invoke $\mathcal{F}_{\text{mult-mal}}$ with input sharings $[\boldsymbol{F}(i)], [\boldsymbol{G}(i)]$ to compute $[\boldsymbol{z}^{(i)}]$. All parties determine $\boldsymbol{H}$ by the following $2m-1$ evaluation points: $\forall i \in \{1, \ldots, 2m-1\}$, $\boldsymbol{H}(i) = \boldsymbol{z}^{(i)}$. Recall that $\mathcal{F}_{\text{coin}}$ introduced in Section 3.3 allows all parties to generate a random element in $\mathbb{F}$. All parties use $\mathcal{F}_{\text{coin}}$ to generate a random evaluation point $\lambda \in \mathbb{F}$ and then take $[\boldsymbol{F}(\lambda)], [\boldsymbol{G}(\lambda)], [\boldsymbol{H}(\lambda)]$ as output. When $\mathbb{F}$ is large enough, it is sufficient to test whether $([\boldsymbol{F}(\lambda)], [\boldsymbol{G}(\lambda)], [\boldsymbol{H}(\lambda)])$ is a correct multiplication tuple.

The description of Compress appears in Protocol 36. The communication complexity of Compress is $O(m \cdot n \cdot \kappa + n^3\kappa^2)$ bits.

Lemma 18. *Let κ be the security parameter and $\mathbb{F}$ be a finite field such that $|\mathbb{F}| \geq 2^\kappa$. Then, with probability at least $1 - 7m/2^\kappa$,*

- *$\lambda \notin [m]$;*
- *if there exists $i \in [m]$ such that $[\boldsymbol{x}^{(i)}]$ is inconsistent, then $[\boldsymbol{F}(\lambda)]$ is also inconsistent;*
- *if there exists $i \in [m]$ such that $[\boldsymbol{y}^{(i)}]$ is inconsistent, then $[\boldsymbol{G}(\lambda)]$ is also inconsistent;*
- *if there exists $i \in [m]$ such that $[\boldsymbol{z}^{(i)}]$ is inconsistent, then $[\boldsymbol{H}(\lambda)]$ is also inconsistent;*
- *if there exists $i \in [m]$ such that $\boldsymbol{z}^{(i)} \neq \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$, then $\boldsymbol{H}(\lambda) \neq \boldsymbol{F}(\lambda) * \boldsymbol{G}(\lambda)$.*

Proof. We count the number of λ that breaks any of the conditions.

- The number of λ that breaks the first condition is m.
- For the second condition, we show that if there exists $i \in [m]$ such that $[\boldsymbol{y}^{(i)}]$ is inconsistent, then the number of λ such that $[\boldsymbol{F}(\lambda)]$ is consistent is bounded by $m-1$. Suppose there are m evaluation points $\lambda_1, \lambda_2, \ldots, \lambda_m$ such that $[\boldsymbol{F}(\lambda_i)]$ is consistent for all $i \in [m]$. Since $\boldsymbol{F}$ is a vector of degree-$(m-1)$ polynomials, the coefficients of $[\boldsymbol{F}(\cdot)]$ are linear combinations of $\{[\boldsymbol{F}(\lambda_i)]\}_{i=1}^m$, which means that every coefficient of $[\boldsymbol{F}(\cdot)]$ is a consistent degree-t packed Shamir sharing. Therefore for all λ, $[\boldsymbol{F}(\lambda)]$ is consistent, which implies that $[\boldsymbol{F}(1)] = [\boldsymbol{x}^{(1)}], \ldots, [\boldsymbol{F}(m)] = [\boldsymbol{x}^{(m)}]$ are all consistent. It leads to a contradiction. Thus, the number of λ such that $[\boldsymbol{F}(\lambda)]$ is consistent is bounded by $m-1$. The number of λ that breaks the second condition is bounded by $m-1$.

Protocol 36: COMPRESS

1. Let m denote the number of multiplication tuples. The multiplication tuples are denoted by
$$([\boldsymbol{x}^{(1)}], [\boldsymbol{y}^{(1)}], [\boldsymbol{z}^{(1)}]), ([\boldsymbol{x}^{(2)}], [\boldsymbol{y}^{(2)}], [\boldsymbol{z}^{(2)}]), \ldots, ([\boldsymbol{x}^{(m)}].[\boldsymbol{y}^{(m)}], [\boldsymbol{z}^{(m)}]).$$
2. Let $\boldsymbol{F}(\cdot), \boldsymbol{G}(\cdot)$ be vectors of degree-$(m-1)$ polynomials such that
$$\forall i \in [m], \boldsymbol{F}(i) = \boldsymbol{x}^{(i)}, \boldsymbol{G}(i) = \boldsymbol{y}^{(i)}.$$
All parties locally compute $[\boldsymbol{F}(\cdot)]$ and $[\boldsymbol{G}(\cdot)]$ by using $\{[\boldsymbol{x}^{(i)}]\}_{i\in[m]}$ and $\{[\boldsymbol{y}^{(i)}]\}_{i\in[m]}$ respectively. Note that the coefficients of $[\boldsymbol{F}(\cdot)]$ and $[\boldsymbol{G}(\cdot)]$ are linear combinations of $\{[\boldsymbol{x}^{(i)}]\}_{i\in[m]}$ and $\{[\boldsymbol{y}^{(i)}]\}_{i\in[m]}$ respectively
3. For all $i \in \{m+1, \ldots, 2m-1\}$, all parties locally compute $[\boldsymbol{F}(i)]$ and $[\boldsymbol{G}(i)]$, and then invoke $\mathcal{F}_{\text{mult-mal}}$ on $([\boldsymbol{F}(i)], [\boldsymbol{G}(i)])$ to compute $[\boldsymbol{z}^{(i)}] = [\boldsymbol{F}(i) * \boldsymbol{G}(i)]$.
4. Let $\boldsymbol{H}(\cdot)$ be a vector of degree-$2(m-1)$ polynomials such that
$$\forall i \in [2m-1], \boldsymbol{H}(i) = \boldsymbol{z}^{(i)}.$$
All parties locally compute $[\boldsymbol{H}(\cdot)]$ by using $\{[\boldsymbol{z}^{(i)}]\}_{i\in[2m-1]}$. Note that the coefficients of $[\boldsymbol{H}(\cdot)]$ are linear combinations of $\{[\boldsymbol{z}^{(i)}]\}_{i\in[2m-1]}$.
5. All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random field element λ. If $\lambda \in [m]$, all parties abort. Otherwise, output $([\boldsymbol{F}(\lambda)], [\boldsymbol{G}(\lambda)], [\boldsymbol{H}(\lambda)])$.

- With the same argument as that for the second condition, the number of λ that breaks the third condition is bounded by $m-1$.
- With the same argument as that for the second condition and the fact that $\boldsymbol{H}$ is a vector of degree-$2(m-1)$ polynomials, the number of λ that breaks the fourth condition is bounded by $2(m-1)$.
- For the last condition, consider a vector of degree-$2(m-1)$ polynomials $\boldsymbol{\delta}(\cdot) = \boldsymbol{H}(\cdot) - \boldsymbol{F}(\cdot) * \boldsymbol{H}(\cdot)$. If there exists $i \in [m]$ such that $\boldsymbol{z}^{(i)} \neq \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$, then $\boldsymbol{\delta}(\cdot)$ is non-zero. Since $\boldsymbol{\delta}$ is a vector of degree-$2(m-1)$ polynomials, the number of λ such that $\boldsymbol{\delta}(\lambda) = \boldsymbol{0}$ is bounded by $2(m-1)$. It means that the number of λ such that $\boldsymbol{H}(\lambda) = \boldsymbol{F}(\lambda) * \boldsymbol{H}(\lambda)$ is bounded by $2(m-1)$. Therefore, the number of λ that breaks the last condition is bounded by $2(m-1)$.

In summary, the number of λ that breaks any of the conditions is bounded by $m + (m-1) + (m-1) + 2(m-1) + 2(m-1) < 7m$. Since λ is a random field element sampled by $\mathcal{F}_{\text{coin}}$, the probability that any of the conditions does not hold is bounded by $7m/|\mathbb{F}| \leq 7m/2^\kappa$. The lemma follows. □

Remark 4. *We note that the technique from [BSFO12] is already a verification for multiplication tuples. However, it inevitably requires a large enough field. While the requirement that $|\mathbb{F}| \geq 2^\kappa$ can be mitigated by using a large enough extension field when choosing a random element λ in the last step of* COMPRESS, *the existence of $\boldsymbol{H}$ requires at least $2m-1$ different evaluation points, which means that $|\mathbb{F}| \geq 2m-1$. Note that we need to invoke additional $O(m)$ times of $\mathcal{F}_{\textit{mult-mal}}$. It will affect the overall communication complexity. There are some discussions in [BSFO12] regarding how to mitigate the requirement of a large enough field. It requires to choose the parameters very carefully. On the other hand, using the technique in [BBCG$^+$19, GSZ20], we can safely lift the multiplication tuples into a large enough extension field without* *affecting the overall communication complexity since the communication complexity of the verification is sub-linear in the number of multiplications.*

7.1.2 Extensions of Mult and Compress.

Before introducing the verification technique from [BBCG+19, GSZ20], we first extend Mult to support inner-product operations and extend Compress to support the verification of inner-product tuples. The idea of extending Mult to support inner-product has been widely used in many papers. The idea of extending Compress to support the verification of inner-product tuples is first noted in [NV18], which is then used in [GSZ20] to construct a more efficient verification. For completeness, we provide the full details below.

An Extension of Mult. Consider that we have two input vectors of sharings $([\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(\ell)}])$ and $([\boldsymbol{y}^{(1)}], [\boldsymbol{y}^{(2)}], \ldots, [\boldsymbol{y}^{(\ell)}])$. The goal is to compute a degree-t packed Shamir sharing of $\boldsymbol{z} = \sum_{i=1}^{\ell} \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$. Note that all parties can locally compute $\langle \boldsymbol{z} \rangle := \sum_{i=1}^{\ell} [\boldsymbol{x}^{(i)}] \cdot [\boldsymbol{y}^{(i)}]$. Then use the approach in [DN07] to reduce the degree of $\langle \boldsymbol{z} \rangle$. The functionality $\mathcal{F}_{\text{inner-product-mal}}$ appears in Functionality 37, and the protocol Inner-product appears in Protocol 38. Note that the communication complexity of Inner-product is the same as the communication complexity of Mult.

Functionality 37: $\mathcal{F}_{\text{inner-product-mal}}$

1. Suppose $([\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(\ell)}])$ and $([\boldsymbol{y}^{(1)}], [\boldsymbol{y}^{(2)}], \ldots, [\boldsymbol{y}^{(\ell)}])$ are the input vectors of degree-t packed Shamir sharings. $\mathcal{F}_{\text{inner-product-mal}}$ receives the shares of $\{[\boldsymbol{x}^{(i)}], [\boldsymbol{y}^{(i)}]\}_{i=1}^{\ell}$ from honest parties.
2. For all $i \in [\ell]$, $\mathcal{F}_{\text{inner-product-mal}}$ recovers the whole sharings $[\boldsymbol{x}^{(i)}]_{\mathcal{H}}, [\boldsymbol{x}^{(i)}]$ and $[\boldsymbol{y}^{(i)}]_{\mathcal{H}}, [\boldsymbol{y}^{(i)}]$. $\mathcal{F}_{\text{inner-product-mal}}$ computes $\boldsymbol{\Delta}_x^{(i)} = [\boldsymbol{x}^{(i)}] - [\boldsymbol{x}^{(i)}]_{\mathcal{H}}$ and $\boldsymbol{\Delta}_y^{(i)} = [\boldsymbol{y}^{(i)}] - [\boldsymbol{y}^{(i)}]_{\mathcal{H}}$. Then $\mathcal{F}_{\text{inner-product-mal}}$ sends the shares of $[\boldsymbol{x}^{(i)}]_{\mathcal{H}}, [\boldsymbol{y}^{(i)}]_{\mathcal{H}}$ of corrupted parties and $\boldsymbol{\Delta}_x^{(i)}, \boldsymbol{\Delta}_y^{(i)}$ to the adversary.
3. $\mathcal{F}_{\text{inner-product-mal}}$ receives from the adversary a vector $\boldsymbol{d} \in \mathbb{F}^k$. $\mathcal{F}_{\text{inner-product-mal}}$ computes $\langle \boldsymbol{z} \rangle := \sum_{i=1}^{\ell} [\boldsymbol{x}^{(i)}] \cdot [\boldsymbol{y}^{(i)}]$ and reconstructs the secrets $\boldsymbol{z}$. $\mathcal{F}_{\text{inner-product-mal}}$ computes $\boldsymbol{z} := \boldsymbol{z} + \boldsymbol{d}$.
4. $\mathcal{F}_{\text{inner-product-mal}}$ receives from the adversary a set of shares $\{s_i\}_{i \in \mathcal{C}}$. $\mathcal{F}_{\text{inner-product-mal}}$ samples a random degree-t packed Shamir sharing $[\boldsymbol{z}]_{\mathcal{H}}$ such that for all $P_i \in \mathcal{C}$, the i-th share of $[\boldsymbol{z}]_{\mathcal{H}}$ is s_i.
5. $\mathcal{F}_{\text{inner-product-mal}}$ receives from the adversary a vector $\boldsymbol{\Delta}_z \in \mathbb{F}^n$ such that for all $P_i \notin \mathcal{H}_{\mathcal{C}}$, $(\boldsymbol{\Delta}_z)_i = 0$. $\mathcal{F}_{\text{inner-product-mal}}$ computes $[\boldsymbol{z}] = [\boldsymbol{z}]_{\mathcal{H}} + \boldsymbol{\Delta}_z$.
6. $\mathcal{F}_{\text{inner-product-mal}}$ distributes the shares of $[\boldsymbol{z}]$ to honest parties.

Lemma 19. *Protocol* Inner-product *(see Protocol 38) securely computes $\mathcal{F}_{\text{inner-product-mal}}$ in the $\mathcal{F}_{\text{rand}}$-hybrid model against a fully malicious adversary who controls $t' = t - k + 1$ parties.*

This lemma can be proved in the same way as that for Lemma 12. Therefore, for simplicity, we omit the details.

An Extension of Compress. Consider the scenario where we want to check the correctness of m inner-product tuples of the same dimension ℓ, where the i-th tuple is represented by

$$(([\boldsymbol{x}^{(i,1)}], [\boldsymbol{x}^{(i,2)}], \ldots, [\boldsymbol{x}^{(i,\ell)}]), ([\boldsymbol{y}^{(i,1)}], [\boldsymbol{y}^{(i,2)}], \ldots, [\boldsymbol{y}^{(i,\ell)}]), [\boldsymbol{z}^{(i)}]).$$

Protocol 38: Inner-product

1. Suppose $([\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(\ell)}])$ and $([\boldsymbol{y}^{(1)}], [\boldsymbol{y}^{(2)}], \ldots, [\boldsymbol{y}^{(\ell)}])$ denote the input vectors of degree-t packed Shamir sharings of the multiplication gate.
2. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare a pair of random sharings $([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$.
3. All parties locally compute $\langle \boldsymbol{e} \rangle := \sum_{i=1}^{\ell} [\boldsymbol{x}^{(i)}] \cdot [\boldsymbol{y}^{(i)}] + \langle \boldsymbol{r} \rangle$.
4. All parties send their shares of $\langle \boldsymbol{e} \rangle$ to the first party P_1.
5. P_1 reconstructs the secrets $\boldsymbol{e}$, generates a random degree-t packed Shamir sharing $[\boldsymbol{e}]$, and distributes the shares to other parties.
6. All parties locally compute $[\boldsymbol{z}] := [\boldsymbol{e}] - [\boldsymbol{r}]$.

At a high-level, all parties will construct $2\ell + 1$ vectors of polynomials

$$(\boldsymbol{F}^{(1)}(\cdot), \boldsymbol{F}^{(2)}(\cdot), \ldots, \boldsymbol{F}^{(\ell)}(\cdot)), (\boldsymbol{G}^{(1)}(\cdot), \boldsymbol{G}^{(2)}(\cdot), \ldots, \boldsymbol{G}^{(\ell)}(\cdot)), \boldsymbol{H}(\cdot),$$

where the coefficients of these polynomials are vectors in $\mathbb{F}^k$, $\boldsymbol{F}^{(j)}, \boldsymbol{G}^{(j)}$ are of degree-$(m-1)$ for all $j \in [\ell]$, and $\boldsymbol{H}$ is of degree-$2(m-1)$. For all $j \in [\ell]$, $\boldsymbol{F}^{(j)}, \boldsymbol{G}^{(j)}$ are determined by the following m evaluation points: $\forall i \in [m]$, $\boldsymbol{F}^{(j)}(i) = \boldsymbol{x}^{(i,j)}$ and $\boldsymbol{G}^{(j)}(i) = \boldsymbol{y}^{(i,j)}$. For $\boldsymbol{H}$, we want to achieve that

- If the input inner-product tuples are correct and all parties honestly follow the protocol, then $\boldsymbol{H} = \sum_{j=1}^{\ell} \boldsymbol{F}^{(j)} * \boldsymbol{G}^{(j)}$.
- Otherwise, $\boldsymbol{H} \neq \sum_{j=1}^{\ell} \boldsymbol{F}^{(j)} * \boldsymbol{G}^{(j)}$.

In this way, it is sufficient to test whether $\boldsymbol{H} = \sum_{j=1}^{\ell} \boldsymbol{F}^{(j)} * \boldsymbol{G}^{(j)}$, which can be done by testing a random evaluation point. To this end, for all $i \in [m]$, $\boldsymbol{H}(i) = \boldsymbol{z}^{(i)}$. Note that we need additional $m-1$ points to determine the polynomial $\boldsymbol{H}$. For all $i \in \{m+1, \ldots, 2m-1\}, j \in [\ell]$, all parties locally compute $[\boldsymbol{F}^{(j)}(i)], [\boldsymbol{G}^{(j)}(i)]$, which are linear combinations of $[\boldsymbol{x}^{(1,j)}], \ldots, [\boldsymbol{x}^{(m,j)}]$ and $[\boldsymbol{y}^{(1,j)}], \ldots, [\boldsymbol{y}^{(m,j)}]$ respectively. Then, all parties invoke $\mathcal{F}_{\text{inner-product-mal}}$ with input sharings $([\boldsymbol{F}^{(1)}(i)], \ldots, [\boldsymbol{F}^{(\ell)}(i)])$ and $([\boldsymbol{G}^{(1)}(i)], \ldots, [\boldsymbol{G}^{(\ell)}(i)])$ to compute $[\boldsymbol{z}^{(i)}]$. All parties determine $\boldsymbol{H}$ by the following $2m-1$ evaluation points: $\forall i \in \{1, \ldots, 2m-1\}$, $\boldsymbol{H}(i) = \boldsymbol{z}^{(i)}$.

Recall that $\mathcal{F}_{\text{coin}}$ introduced in Section 3.3 allows all parties to generate a random element in $\mathbb{F}$. All parties use $\mathcal{F}_{\text{coin}}$ to generate a random evaluation point $\lambda \in \mathbb{F}$ and then take

$$([\boldsymbol{F}^{(1)}(\lambda)], \ldots, \boldsymbol{F}^{(\ell)}(\lambda)), ([\boldsymbol{G}^{(1)}(\lambda)], \ldots, \boldsymbol{G}^{(\ell)}(\lambda)), [\boldsymbol{H}(\lambda)]$$

as output. When $\mathbb{F}$ is large enough, it is sufficient to test whether

$$(([\boldsymbol{F}^{(1)}(\lambda)], \ldots, \boldsymbol{F}^{(\ell)}(\lambda)), ([\boldsymbol{G}^{(1)}(\lambda)], \ldots, \boldsymbol{G}^{(\ell)}(\lambda)), [\boldsymbol{H}(\lambda)])$$

is a correct multiplication tuple.

The description of Extend-Compress appears in Protocol 39. The communication complexity of Extend-Compress is $O(m \cdot n \cdot \kappa + n^3 \cdot \kappa^2)$ bits.

Lemma 20. *Let κ be the security parameter and $\mathbb{F}$ be a finite field such that $|\mathbb{F}| \geq 2^\kappa$. Then, with probability at least $1 - (2\ell + 5)m/2^\kappa$,*

- $\lambda \notin [m]$;

Protocol 39: EXTEND-COMPRESS

1. Let m denote the number of inner-product tuples and ℓ denote the dimension of each inner-product tuple. For all $i \in [m]$, the i-th inner-product tuple is denoted by
$$(([\boldsymbol{x}^{(i,1)}], [\boldsymbol{x}^{(i,2)}], \ldots, [\boldsymbol{x}^{(i,\ell)}]), ([\boldsymbol{y}^{(i,1)}], [\boldsymbol{y}^{(i,2)}], \ldots, [\boldsymbol{y}^{(i,\ell)}]), [\boldsymbol{z}^{(i)}]).$$
2. For all $j \in [\ell]$, let $\boldsymbol{F}^{(j)}(\cdot), \boldsymbol{G}^{(j)}(\cdot)$ be vectors of degree-$(m-1)$ polynomials such that
$$\forall i \in [m], \boldsymbol{F}^{(j)}(i) = \boldsymbol{x}^{(i,j)}, \boldsymbol{G}^{(j)}(i) = \boldsymbol{y}^{(i,j)}.$$
All parties locally compute $[\boldsymbol{F}^{(j)}(\cdot)]$ and $[\boldsymbol{G}^{(j)}(\cdot)]$ by using $\{[\boldsymbol{x}^{(i,j)}]\}_{i\in[m]}$ and $\{[\boldsymbol{y}^{(i,j)}]\}_{i\in[m]}$ respectively. Note that the coefficients of $[\boldsymbol{F}^{(j)}(\cdot)]$ and $[\boldsymbol{G}^{(j)}(\cdot)]$ are linear combinations of $\{[\boldsymbol{x}^{(i,j)}]\}_{i\in[m]}$ and $\{[\boldsymbol{y}^{(i,j)}]\}_{i\in[m]}$ respectively
3. For all $i \in \{m+1, \ldots, 2m-1\}$, all parties locally compute $[\boldsymbol{F}^{(j)}(i)]$ and $[\boldsymbol{G}^{(j)}(i)]$, and then invoke $\mathcal{F}_{\text{inner-product-mal}}$ on $([\boldsymbol{F}^{(1)}(i)], \ldots, [\boldsymbol{F}^{(m)}(i)]), ([\boldsymbol{G}^{(1)}(i)], \ldots, [\boldsymbol{G}^{(m)}(i)])$ to compute $[\boldsymbol{z}^{(i)}] = [\sum_{j=1}^{\ell} \boldsymbol{F}^{(j)}(i) * \boldsymbol{G}^{(j)}(i)]$.
4. Let $\boldsymbol{H}(\cdot)$ be a vector of degree-$2(m-1)$ polynomials such that
$$\forall i \in [2m-1], \boldsymbol{H}(i) = \boldsymbol{z}^{(i)}.$$
All parties locally compute $[\boldsymbol{H}(\cdot)]$ by using $\{[\boldsymbol{z}^{(i)}]\}_{i\in[2m-1]}$. Note that the coefficients of $[\boldsymbol{H}(\cdot)]$ are linear combinations of $\{[\boldsymbol{z}^{(i)}]\}_{i\in[2m-1]}$.
5. All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random field element λ. If $\lambda \in [m]$, all parties abort. Otherwise, output
$$(([\boldsymbol{F}^{(1)}(\lambda)], \ldots, \boldsymbol{F}^{(\ell)}(\lambda)), ([\boldsymbol{G}^{(1)}(\lambda)], \ldots, \boldsymbol{G}^{(\ell)}(\lambda)), [\boldsymbol{H}(\lambda)]).$$

- *for all $j \in [\ell]$, if there exists $i \in [m]$ such that $[\boldsymbol{x}^{(i,j)}]$ is inconsistent, then $[\boldsymbol{F}^{(j)}(\lambda)]$ is also inconsistent;*
- *for all $j \in [\ell]$, if there exists $i \in [m]$ such that $[\boldsymbol{y}^{(i,j)}]$ is inconsistent, then $[\boldsymbol{G}^{(j)}(\lambda)]$ is also inconsistent;*
- *if there exists $i \in [m]$ such that $[\boldsymbol{z}^{(i)}]$ is inconsistent, then $[\boldsymbol{H}(\lambda)]$ is also inconsistent;*
- *if there exists $i \in [m]$ such that $\boldsymbol{z}^{(i)} \neq \sum_{j=1}^{\ell} \boldsymbol{x}^{(i,j)} * \boldsymbol{y}^{(i,j)}$, then $\boldsymbol{H}(\lambda) \neq \sum_{j=1}^{\ell} \boldsymbol{F}^{(j)}(\lambda) * \boldsymbol{G}^{(j)}(\lambda)$.*

Proof. We count the number of λ that breaks any of the conditions.

- The number of λ that breaks the first condition is m.
- For the second condition, following the same argument as that in Lemma 18, for each fixed j, the number of λ that breaks the second condition is bounded by $m-1$. Therefore, the total number of λ that breaks the second condition is bounded by $\ell(m-1)$.
- With the same argument as that for the second condition, the number of λ that breaks the third condition is bounded by $\ell(m-1)$.
- With the same argument as that in Lemma 18, the number of λ that breaks the fourth condition is bounded by $2(m-1)$.

- For the last condition, consider a vector of degree-$2(m-1)$ polynomials $\boldsymbol{\delta}(\cdot) = \boldsymbol{H}(\cdot) - \sum_{j=1}^{\ell} \boldsymbol{F}^{(j)}(\cdot) * \boldsymbol{H}^{(j)}(\cdot)$. If there exists $i \in [m]$ such that $\boldsymbol{z}^{(i)} \neq \sum_{j=1}^{\ell} \boldsymbol{x}^{(i,j)} * \boldsymbol{y}^{(i,j)}$, then $\boldsymbol{\delta}(\cdot)$ is non-zero. Since $\boldsymbol{\delta}$ is a vector of degree-$2(m-1)$ polynomials, the number of λ such that $\boldsymbol{\delta}(\lambda) = \boldsymbol{0}$ is bounded by $2(m-1)$. It means that the number of λ such that $\boldsymbol{H}(\lambda) = \sum_{j=1}^{\ell} \boldsymbol{F}^{(j)}(\lambda) * \boldsymbol{H}^{(j)}(\lambda)$ is bounded by $2(m-1)$. Therefore, the number of λ that breaks the last condition is bounded by $2(m-1)$.

In summary, the number of λ that breaks any of the conditions is bounded by $m + \ell(m-1) + \ell(m-1) + 2(m-1) + 2(m-1) < (2\ell+5)m$. Since λ is a random field element sampled by $\mathcal{F}_{\text{coin}}$, the probability that any of the conditions does not hold is bounded by $(2\ell+5)m/|\mathbb{F}| \leq (2\ell+5)m/2^{\kappa}$. The lemma follows. $\square$

7.1.3 Multiplication Verification from [GSZ20].

Now we are ready to discuss how to use the technique in [BBCG$^+$19, GSZ20] to verify multiplication tuples. We will follow the approach in [GSZ20]. At a high-level, the verification is done by the following three steps:

1. De-Linearization: The first step is to transform the m input multiplication tuples into a single inner-product tuple of dimension m such that if there exists an incorrect multiplication tuple, the inner-product tuple is also incorrect with overwhelming probability.

2. Dimension-Reduction: The second step is to reduce the dimension of the inner-product tuple we obtained in Step 1 by a factor of κ, where κ is the security parameter. More concretely, taking as input an inner-product tuple of dimension m, the output is an inner-product tuple of dimension m/κ such that if the input inner-product tuple is incorrect, the output inner-product tuple is also incorrect with overwhelming probability.

3. Recursion and Randomization: We will repeat the second step until the dimension becomes 1, i.e., the final output becomes a single multiplication tuple. To simplify the checking process for the last step, the last iteration will use additional randomness.

In the following, we will lift all the sharings into a large enough extension field of $\mathbb{F}$, denoted by $\mathbb{K}$, such that $|\mathbb{K}| \geq 2^{\kappa}$. We assume that κ is the size of a field element in $\mathbb{K}$.

Step 1: De-Linearization. The first step is to transform these m tuples to a single inner-product tuple of dimension m. The description of De-Linearization appears in Protocol 40. The communication complexity of De-Linearization is $O(n^2 \cdot \kappa)$ bits.

Lemma 21. *In* De-Linearization, *with probability at least* $1 - 2m/2^{\kappa}$,

- *if there exists* $i \in [m]$ *such that* $[\boldsymbol{z}^{(i)}]$ *is inconsistent, then* $[\boldsymbol{c}]$ *is also inconsistent;*
- *if there exists* $i \in [m]$ *such that* $\boldsymbol{z}^{(i)} \neq \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$, *then* $\boldsymbol{c} \neq \sum_{i=1}^{m} \boldsymbol{a}^{(i)} * \boldsymbol{b}^{(i)}$.

Proof. Consider two vectors of polynomials $\boldsymbol{F}, \boldsymbol{G}$ defined by

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

Note that the inner-product between $(\boldsymbol{a}^{(1)}, \boldsymbol{a}^{(2)}, \ldots, \boldsymbol{a}^{(m)})$ and $(\boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \ldots, \boldsymbol{b}^{(m)})$ is $\boldsymbol{F}(r)$, and $\boldsymbol{c} = \boldsymbol{G}(r)$.

We count the number of r that breaks either of the conditions.

- For the first condition, consider the polynomial

$$[\boldsymbol{G}(X)] = [\boldsymbol{z}^{(1)}] + [\boldsymbol{z}^{(2)}] \cdot X + \ldots + [\boldsymbol{z}^{(m)}] \cdot X^{m-1}.$$

Suppose there are m evaluation points $r_1, r_2, \ldots, r_m$ such that $[\boldsymbol{G}(r_i)]$ is consistent for all $i \in [m]$. Since $\boldsymbol{G}$ is a vector of degree-$(m-1)$ polynomials, the coefficients of $[\boldsymbol{G}(\cdot)]$ are linear combinations

Protocol 40: De-Linearization

1. Let m denote the number of multiplication tuples. The multiplication tuples are denoted by
$$([\boldsymbol{x}^{(1)}], [\boldsymbol{y}^{(1)}], [\boldsymbol{z}^{(1)}]), ([\boldsymbol{x}^{(2)}], [\boldsymbol{y}^{(2)}], [\boldsymbol{z}^{(2)}]), \ldots, ([\boldsymbol{x}^{(m)}].[\boldsymbol{y}^{(m)}], [\boldsymbol{z}^{(m)}]).$$
2. All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random field element $r \in \mathbb{K}$.
3. For all $i \in [m]$, all parties set $[\boldsymbol{a}^{(i)}] = r^{i-1} \cdot [\boldsymbol{x}^{(i)}]$ and set $[\boldsymbol{b}^{(i)}] = [\boldsymbol{y}^{(i)}]$. All parties compute
$$[\boldsymbol{c}] = \sum_{i=1}^{m} r^{i-1} \cdot [\boldsymbol{z}^{(i)}]$$
and output the inner-product tuple
$$(([\boldsymbol{a}^{(1)}], [\boldsymbol{a}^{(2)}], \ldots, [\boldsymbol{a}^{(m)}]), ([\boldsymbol{b}^{(1)}], [\boldsymbol{b}^{(2)}], \ldots, [\boldsymbol{b}]^{(m)}), [\boldsymbol{c}]).$$

of $\{[\boldsymbol{G}(r_i)]\}_{i=1}^{m}$, which means that every coefficient of $[\boldsymbol{G}(\cdot)]$ is a consistent degree-t packed Shamir sharing. Therefore $[\boldsymbol{z}^{(1)}], \ldots, [\boldsymbol{z}^{(m)}]$ are all consistent. It means that if there exists $i \in [m]$ such that $[\boldsymbol{z}^{(i)}]$ is inconsistent, then the number of r such that $[\boldsymbol{G}(r)]$ is consistent is bounded by $m-1$. Thus, the number of r that breaks the first condition is bounded by $m-1$.

- For the second condition, consider a vector of degree-$m-1$ polynomials $\boldsymbol{\delta}(\cdot) = \boldsymbol{G}(\cdot) - \boldsymbol{F}(\cdot)$. If there exists $i \in [m]$ such that $\boldsymbol{z}^{(i)} \neq \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$, then $\boldsymbol{\delta}(\cdot)$ is non-zero. Since $\boldsymbol{\delta}$ is a vector of degree-$m-1$ polynomials, the number of r such that $\boldsymbol{\delta}(r) = \boldsymbol{0}$ is bounded by $m-1$. It means that the number of r such that $\boldsymbol{c} = \sum_{i=1}^{m} \boldsymbol{a}^{(i)} * \boldsymbol{b}^{(i)}$ is bounded by $m-1$. Therefore, the number of r that breaks the second condition is bounded by $m-1$.

In summary, the number of r that breaks any of the conditions is bounded by $(m-1)+(m-1) < 2m$. Since r is a random field element sampled by $\mathcal{F}_{\text{coin}}$, the probability that either of the conditions does not hold is bounded by $2m/|\mathbb{K}| \leq 2m/2^{\kappa}$. The lemma follows. □

Step 2: Dimension-Reduction. The second step is to reduce the dimension of the inner-product tuple we obtained in Step 1 by a factor of κ, where κ is the security parameter. Let m denote the dimension of the input inner-product tuple and $\ell = m/\kappa$. At a high-level, all parties first partition the two input vectors of the inner-product tuple into κ sub-vectors of the same length ℓ. For all $i \in [\kappa]$, all parties invoke $\mathcal{F}_{\text{inner-product-mal}}$ to compute the inner-product between the i-th sub-vectors of the two input vectors of the original inner-product tuple. In this way, all parties obtain κ inner-product tuples of dimension ℓ. Finally all parties invoke Extend-Compress to compress the check of these κ inner-product tuples into one check of a single inner-product tuple. The description of Dimension-Reduction appears in Protocol 41. The communication complexity of Dimension-Reduction is $O(n^3 \cdot \kappa^2)$ bits.

Lemma 22. *In* Dimension-Reduction, *with probability at least* $1 - (2\ell + 5)m/2^{\kappa}$,

- *if there exists* $j \in [m]$ *such that* $[\boldsymbol{x}^{(j)}]$ *is inconsistent, then there exists* $j' \in [\ell]$ *such that* $[\boldsymbol{a}^{(j')}]$ *is inconsistent;*
- *if there exists* $j \in [m]$ *such that* $[\boldsymbol{y}^{(j)}]$ *is inconsistent, then there exists* $j' \in [\ell]$ *such that* $[\boldsymbol{b}^{(j')}]$ *is inconsistent;*
- *if* $[\boldsymbol{z}]$ *is inconsistent, then* $[\boldsymbol{c}]$ *is inconsistent;*

Protocol 41: Dimension-Reduction

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

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

   Recall that κ is the security parameter. Let $\ell = m/\kappa$.

2. For all $i \in [\kappa], j \in [\ell]$, all parties set $[\boldsymbol{a}^{(i,j)}] := [\boldsymbol{x}^{((i-1)\cdot\ell+j)}]$ and $[\boldsymbol{b}^{(i,j)}] := [\boldsymbol{y}^{((i-1)\cdot\ell+j)}]$.

3. For $i \in [\kappa - 1]$, all parties invoke $\mathcal{F}_{\text{inner-product-mal}}$ on $([\boldsymbol{a}^{(i,1)}], \dots, [\boldsymbol{a}^{(i,\ell)}]), ([\boldsymbol{b}^{(i,1)}], \dots, [\boldsymbol{b}^{(i,\ell)}])$ to compute $[\boldsymbol{c}^{(i)}] = [\sum_{j=1}^{\ell} \boldsymbol{a}^{(i,j)} * \boldsymbol{b}^{(i,j)}]$

4. All parties set

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

5. All parties invoke Extend-Compress on the following κ inner-product tuples:

$$\{(([\boldsymbol{a}^{(i,1)}], \dots, [\boldsymbol{a}^{(i,\ell)}]), ([\boldsymbol{b}^{(i,1)}], \dots, [\boldsymbol{b}^{(i,\ell)}]), [\boldsymbol{c}^{(i)}])\}_{i=1}^{\kappa}.$$

   The output is denoted by

$$(([\boldsymbol{a}^{(1)}], \dots, [\boldsymbol{a}^{(\ell)}]), ([\boldsymbol{b}^{(1)}], \dots, [\boldsymbol{b}^{(\ell)}]), [\boldsymbol{c}]).$$

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

- *if* $\boldsymbol{z} \neq \sum_{j=1}^{m} \boldsymbol{x}^{(j)} * \boldsymbol{y}^{(j)}$, *then* $\boldsymbol{c} \neq \sum_{j=1}^{\ell} \boldsymbol{a}^{(j)} * \boldsymbol{b}^{(j)}$.

Proof. We first show that, if $[\boldsymbol{z}]$ is inconsistent, then there exists $i' \in [\kappa]$ such that $[\boldsymbol{c}^{(i')}]$ is inconsistent. Note that in Dimension-Reduction, we have

$$[\boldsymbol{z}] = \sum_{i=1}^{\kappa} [\boldsymbol{c}^{(i)}].$$

If for all $i \in [\kappa]$, $[\boldsymbol{c}^{(i)}]$ is consistent, then $[\boldsymbol{z}]$ is also consistent, which leads to a contradiction.

We then show that, if $\boldsymbol{z} \neq \sum_{i=1}^{m} \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$, then there exists $i' \in [\kappa]$ such that $\boldsymbol{c}^{(i')} \neq \sum_{j=1}^{\ell} \boldsymbol{a}^{(i',j)} * \boldsymbol{b}^{(i',j)}$. Note that

$$\sum_{j=1}^{m} \boldsymbol{x}^{(j)} * \boldsymbol{y}^{(j)} = \sum_{i=1}^{\kappa} \sum_{j=1}^{\ell} \boldsymbol{a}^{(i,j)} * \boldsymbol{b}^{(i,j)},$$

and

$$\boldsymbol{z} = \sum_{i=1}^{\kappa} \boldsymbol{c}^{(i)}.$$

If for all $i \in [\kappa]$, $\boldsymbol{c}^{(i)} = \sum_{j=1}^{\ell} \boldsymbol{a}^{(i,j)} * \boldsymbol{b}^{(i,j)}$, then

$$\sum_{i=1}^{\kappa} \boldsymbol{c}^{(i)} = \sum_{i=1}^{\kappa} \sum_{j=1}^{\ell} \boldsymbol{a}^{(i,j)} * \boldsymbol{b}^{(i,j)} = \sum_{j=1}^{m} \boldsymbol{x}^{(j)} * \boldsymbol{y}^{(j)} = \boldsymbol{z},$$

which leads to a contradiction.

Note that the first condition corresponds to the second condition in Lemma 20. The second condition corresponds to the third condition in Lemma 20. The third condition, after transforming $[\boldsymbol{z}]$ being inconsistent to the existence of $i' \in [\kappa]$ such that $[\boldsymbol{c}^{(i')}]$ is inconsistent, corresponds to the fourth condition in Lemma 20. The fourth condition, after transforming $\boldsymbol{z} \neq \sum_{j=1}^{m} \boldsymbol{x}^{(j)} * \boldsymbol{y}^{(j)}$ to the existence of $i' \in [\kappa]$ such that $\boldsymbol{c}^{(i')} \neq \sum_{j=1}^{\ell} \boldsymbol{a}^{(i',j)} * \boldsymbol{b}^{(i',j)}$, corresponds to the last condition in Lemma 20. Thus, by Lemma 20, with probability at least $1 - (2\ell + 5)m/2^{\kappa}$, all conditions hold. $\square$

Step 3: Randomization In the last step, we make use of additional randomness to simplify the verification of the final multiplication tuple. Specifically, in the last call of the second step, we need to compress the check of κ multiplication tuples into a check of a single multiplication tuple. All parties prepare a random multiplication tuple as masks and include it when invoking COMPRESS.

Suppose $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$ is the final multiplication tuple we want to check. In [GSZ20], all parties simply open these three sharings and check whether it is a correct multiplication tuple. Note that opening all three sharings will leak the difference $\boldsymbol{c} - \boldsymbol{a} * \boldsymbol{b}$ to the adversary. As we discussed at the beginning of this subsection, we need to ensure that the input sharings of all the invocations of MULT and INNER-PRODUCT are consistent before leaking the difference. Fortunately, we will show that it is sufficient to only check the consistency of the sharings $[\boldsymbol{a}], [\boldsymbol{b}]$. Therefore, we make the following changes when checking the final multiplication tuple:

1. All parties exchange their shares of $[\boldsymbol{a}], [\boldsymbol{b}]$ and verify the consistency of $[\boldsymbol{a}], [\boldsymbol{b}]$. If one of $[\boldsymbol{a}], [\boldsymbol{b}]$ is inconsistent, all parties abort.

2. All parties then exchange their shares of $[\boldsymbol{z}]$ and verify the multiplication tuple. If $[\boldsymbol{c}]$ is inconsistent or $\boldsymbol{c} \neq \boldsymbol{a} * \boldsymbol{b}$, all parties abort.

The description of RANDOMIZATION appears in Protocol 42. The communication complexity of RANDOMIZATION is $O(n^3 \cdot \kappa^2)$ bits.

Lemma 23. *In* RANDOMIZATION*, if the input inner-product tuple is incorrect or any sharing is not consistent, with probability at least $1 - 7m/2^{\kappa}$, at least one honest party aborts at the end of the protocol.*

Proof. Following from the same argument as that in Lemma 22, we have that

- if $[\boldsymbol{z}]$ is inconsistent, then there exists $i' \in [\kappa]$ such that $[\boldsymbol{z}^{(i')}]$ is inconsistent;
- if $\boldsymbol{z} \neq \sum_{i=1}^{\kappa} \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$, then there exists $i' \in [\kappa]$ such that $\boldsymbol{z}^{(i')} \neq \boldsymbol{x}^{(i')} * \boldsymbol{y}^{(i')}$.

Therefore, if the input inner-product tuple is incorrect or any sharing is not consistent, at least one of the input multiplication tuples of COMPRESS is incorrect or at least one sharing in these input multiplication tuples is inconsistent. By Lemma 18, with probability $1 - 7m/2^{\kappa}$, the output multiplication tuple $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$ is either incorrect or at least one of $[\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}]$ is inconsistent. In either case, at least one honest party aborts in the last two steps of RANDOMIZATION. $\square$

Main Protocol for $\mathcal{F}_{\text{mult-veri}}$. We present the protocol MULT-VERI in Protocol 43, which is a combination of DE-LINEARIZATION, DIMENSION-REDUCTION, and RANDOMIZATION.

Lemma 24. *Protocol* MULT-VERI *(see Protocol 43) securely computes $\mathcal{F}_{\textit{mult-veri}}$ in the $(\mathcal{F}_{\textit{rand}}, \mathcal{F}_{\textit{mult-mal}}, \mathcal{F}_{\textit{inner-product-mal}})$-hybrid model against a fully malicious adversary who controls $t' = t - k + 1$ parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that $\mathcal{H}_{\mathcal{H}} \subset \mathcal{H}$ is a fixed subset of size $t + 1$.

Protocol 42: Randomization

1. Suppose the input inner-product tuple is denoted by
$$(([\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(\kappa)}]), ([\boldsymbol{y}^{(1)}], [\boldsymbol{y}^{(2)}], \ldots, [\boldsymbol{y}]^{(\kappa)}), [\boldsymbol{z}]).$$
Recall that κ is the security parameter.
2. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare two random degree-t packed Shamir sharings $[\boldsymbol{x}^{(0)}], [\boldsymbol{y}^{(0)}]$.
3. All parties invoke $\mathcal{F}_{\text{mult-mal}}$ on $([\boldsymbol{x}^{(0)}], [\boldsymbol{y}^{(0)}])$ to compute $[\boldsymbol{z}^{(0)}] = [\boldsymbol{x}^{(0)} * \boldsymbol{y}^{(0)}]$.
4. For all $i \in [\kappa - 1]$, all parties invoke $\mathcal{F}_{\text{mult-mal}}$ on $([\boldsymbol{x}^{(i)}], [\boldsymbol{y}^{(i)}])$ to compute $[\boldsymbol{z}^{(i)}] = [\boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}]$. Then set
$$[\boldsymbol{z}^{(\kappa)}] = [\boldsymbol{z}] - \sum_{i=1}^{\kappa-1} [\boldsymbol{z}^{(i)}].$$
5. All parties invoke Compress on
$$([\boldsymbol{x}^{(0)}], [\boldsymbol{y}^{(0)}], [\boldsymbol{z}^{(0)}]), ([\boldsymbol{x}^{(1)}], [\boldsymbol{y}^{(1)}], [\boldsymbol{z}^{(1)}]), \ldots, ([\boldsymbol{x}^{(\kappa)}], [\boldsymbol{y}^{(\kappa)}], [\boldsymbol{z}^{(\kappa)}]).$$
The output is denoted by $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$.
6. All parties send their shares of $[\boldsymbol{a}], [\boldsymbol{b}]$ to all other parties. Each party P_i checks whether the shares of $[\boldsymbol{a}], [\boldsymbol{b}]$ it received form valid degree-t packed Shamir sharings. If not, P_i aborts.
7. All parties send their shares of $[\boldsymbol{c}]$ to all other parties. Each party P_i checks whether the shares of $[\boldsymbol{c}]$ it received form valid degree-t packed Shamir sharing and $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. If not, P_i aborts.

Simulation for Mult-veri. In the beginning, for all $i \in [m]$, $\mathcal{S}$ receives from $\mathcal{F}_{\text{mult-veri}}$ the shares of $[\boldsymbol{x}^{(i)}], [\boldsymbol{y}^{(i)}], [\boldsymbol{z}^{(i)}]$ held by corrupted parties and the additive errors $\boldsymbol{\Delta}_x^{(i)}, \boldsymbol{\Delta}_y^{(i)}, \boldsymbol{\Delta}_z^{(i)}$ to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. If $\boldsymbol{\Delta}_x^{(i)} = \boldsymbol{\Delta}_y^{(i)} = \boldsymbol{0}$, $\mathcal{S}$ also receives the difference $\boldsymbol{d}^{(i)} = \boldsymbol{z}^{(i)} - \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$. Furthermore, $\mathcal{S}$ receives $b \in \{\texttt{abort}, \texttt{accept}\}$ indicating whether the multiplications are correct.

- 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 $r \in \mathbb{K}$. The output inner-product tuple is
$$(([\boldsymbol{a}^{(1)}], [\boldsymbol{a}^{(2)}], \ldots, [\boldsymbol{a}^{(m)}]), ([\boldsymbol{b}^{(1)}], [\boldsymbol{b}^{(2)}], \ldots, [\boldsymbol{b}]^{(m)}), [\boldsymbol{c}]).$$
  For each sharing, $\mathcal{S}$ computes the shares held by corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. If $\mathcal{S}$ receives from $\mathcal{F}_{\text{mult-veri}}$ all the additive errors to the multiplication results of the input multiplication tuples, $\mathcal{S}$ also computes the additive errors to the inner-product results of the output inner-product tuple. More concretely,

  1. For all $i \in [m]$, since $[\boldsymbol{a}^{(i)}] = r^{i-1} \cdot [\boldsymbol{x}^{(i)}]$, the shares of $[\boldsymbol{a}^{(i)}]$ held by corrupted parties can be computed by using the shares of $[\boldsymbol{x}^{(i)}]$ held by corrupted parties. The additive errors to the shares of $[\boldsymbol{a}^{(i)}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ are $\boldsymbol{\Delta}_a^{(i)} = r^{(i-1)} \cdot \boldsymbol{\Delta}_x^{(i)}$.
  2. For all $i \in [m]$, since $[\boldsymbol{b}^{(i)}] = [\boldsymbol{y}^{(i)}]$, the shares of $[\boldsymbol{b}^{(i)}]$ held by corrupted parties are the same as the shares of $[\boldsymbol{y}^{(i)}]$ held by corrupted parties. The additive errors to the shares of $[\boldsymbol{b}^{(i)}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ are $\boldsymbol{\Delta}_b^{(i)} = \boldsymbol{\Delta}_y^{(i)}$.

Protocol 43: Mult-veri

1. Recall that κ is the security parameter. The multiplication tuples are denoted by

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

2. All parties invoke De-Linearization on these m multiplication tuples. Let

$$(([\boldsymbol{a}^{(1)}],\ldots,[\boldsymbol{a}^{(m)}]),([\boldsymbol{b}^{(1)}],\ldots,[\boldsymbol{b}^{(m)}]),[\boldsymbol{c}])$$

denote the output.

3. Let ℓ denote the dimension of the inner-product tuple. Initially $\ell = m$. While $\ell \geq \kappa$, 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

   $$(([\boldsymbol{a}^{(1)}],\ldots,[\boldsymbol{a}^{(\ell/\kappa)}]),([\boldsymbol{b}^{(1)}],\ldots,[\boldsymbol{b}^{(\ell/\kappa)}]),[\boldsymbol{c}]).$$

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

4. Let

$$(([\boldsymbol{a}^{(1)}],\ldots,[\boldsymbol{a}^{(\kappa)}]),([\boldsymbol{b}^{(1)}],\ldots,[\boldsymbol{b}^{(\kappa)}]),[\boldsymbol{c}])$$

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

3. Since $[\boldsymbol{c}] = \sum_{i=1}^{m} r^{i-1}\cdot[\boldsymbol{z}^{(i)}]$, the shares of $[\boldsymbol{c}]$ held by corrupted parties can be computed using the shares of $\{[\boldsymbol{z}^{(i)}]\}_{i=1}^{m}$ held by corrupted parties. The additive errors to the shares of $[\boldsymbol{c}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ are $\boldsymbol{\Delta}_c = \sum_{i=1}^{\ell} r^{i-1}\cdot\boldsymbol{\Delta}_z^{(i)}$. The additive errors to the inner-product results of the output inner-product tuple are $\boldsymbol{d} = \sum_{i=1}^{m} r^{i-1}\cdot\boldsymbol{d}^{(i)}$.

Note that for all $i\in[m]$, if $[\boldsymbol{x}^{(i)}]$ is consistent, then $[\boldsymbol{a}^{(i)}] = r^{i-1}\cdot[\boldsymbol{x}^{(i)}]$ is also consistent. For all $i\in[m]$, if $[\boldsymbol{y}^{(i)}]$ is consistent, then $[\boldsymbol{b}^{(i)}] = [\boldsymbol{y}^{(i)}]$ is also consistent. If either of the following conditions holds, $\mathcal{S}$ aborts:

– if $\boldsymbol{\Delta}_c = \boldsymbol{0}$ but there exists $i\in[m]$ such that $\boldsymbol{\Delta}_z^{(i)} \neq \boldsymbol{0}$;

– if $\boldsymbol{d} = \boldsymbol{0}$ but there exists $i\in[m]$ such that $\boldsymbol{d}^{(i)} \neq \boldsymbol{0}$.

Note that these two conditions correspond to the two conditions in Lemma 21 respectively. According to Lemma 21, it only happens with negligible probability.

- Simulation of Dimension-Reduction:

  We will maintain the invariant that, for the input inner-product tuple

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

  $\mathcal{S}$ learns the shares held by corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ which are denoted by $\{\boldsymbol{\Delta}_x^{(i)}\}_{i=1}^{m},\{\boldsymbol{\Delta}_y^{(i)}\}_{i=1}^{m},\boldsymbol{\Delta}_z$ respectively. In addition, if $\boldsymbol{\Delta}_x^{(i)} = \boldsymbol{\Delta}_y^{(i)} = \boldsymbol{0}$ for all $i\in[m]$, then $\mathcal{S}$ learns the additive errors to the inner-product results, i.e., $\boldsymbol{d} = \boldsymbol{z} - \sum_{i=1}^{m}\boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$. Note that this is true for the first invocation of DIMENSION-REDUCTION since these values are computed when simulating DE-LINEARIZATION.

In Step 2, for all $i \in [\kappa], j \in [\ell]$, $\mathcal{S}$ computes the shares of $[\boldsymbol{a}^{(i,j)}], [\boldsymbol{b}^{(i,j)}]$ held by corrupted parties, and the additive errors to the shares of $[\boldsymbol{a}^{(i,j)}], [\boldsymbol{b}^{(i,j)}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ which are denoted by $\boldsymbol{\Delta}_a^{(i,j)}, \boldsymbol{\Delta}_b^{(i,j)}$. Note that they are just the same as those of $[\boldsymbol{x}^{((i-1)\cdot\ell+j)}], [\boldsymbol{y}^{((i-1)\cdot\ell+j)}]$.

In Step 3, for all $i \in [\kappa - 1]$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{inner-product-mal}}$ when computing $[\boldsymbol{c}^{(i)}]$. For all $j \in [\ell]$, $\mathcal{S}$ sends to the adversary the shares of $[\boldsymbol{a}^{(i,j)}], [\boldsymbol{b}^{(i,j)}]$ held by corrupted parties and $\boldsymbol{\Delta}_a^{(i,j)}, \boldsymbol{\Delta}_b^{(i,j)}$. Then $\mathcal{S}$ receives the shares of $[\boldsymbol{c}^{(i)}]$ held by corrupted parties and the additive errors $\boldsymbol{\Delta}_c^{(i)}$ to the shares of $[\boldsymbol{c}^{(i)}]$ of parties in $\mathcal{H}_{\mathcal{C}}$. $\mathcal{S}$ also receives additive errors $\boldsymbol{d}^{(i)}$ to the secrets of $[\boldsymbol{c}^{(i)}]$. Note that if for all $j \in [\ell]$, $\boldsymbol{\Delta}_x^{(i,j)} = \boldsymbol{\Delta}_y^{(i,j)} = \boldsymbol{0}$, then $\boldsymbol{d}^{(i)} = \boldsymbol{c}^{(i)} - \sum_{j=1}^{\ell} \boldsymbol{a}^{(i,j)} * \boldsymbol{b}^{(i,j)}$.

In Step 4, $\mathcal{S}$ computes the shares of $[\boldsymbol{c}^{(\kappa)}]$ of corrupted parties by following the protocol, and the additive errors $\boldsymbol{\Delta}_c^{(\kappa)} = \boldsymbol{\Delta}_z - \sum_{i=1}^{\kappa-1} \boldsymbol{\Delta}_c^{(i)}$. Since $\boldsymbol{\Delta}_z = \sum_{i=1}^{\kappa} \boldsymbol{\Delta}_c^{(i)}$, if $\boldsymbol{\Delta}_z \neq \boldsymbol{0}$, then there exists $i \in [\kappa]$ such that $\boldsymbol{\Delta}_c^{(i)} \neq \boldsymbol{0}$, which means that $[\boldsymbol{c}^{(i)}]$ is inconsistent. If $\boldsymbol{\Delta}_x^{(i)} = \boldsymbol{\Delta}_y^{(i)} = \boldsymbol{0}$ for all $i \in [m]$, $\mathcal{S}$ computes $\boldsymbol{d}^{(\kappa)} = \boldsymbol{d} - \sum_{i=1}^{\kappa-1} \boldsymbol{d}^{(i)}$. Note that in this case, $\boldsymbol{d}^{(\kappa)} = \boldsymbol{c}^{(\kappa)} - \sum_{j=1}^{\ell} \boldsymbol{a}^{(i,j)} * \boldsymbol{b}^{(i,j)}$.

In Step 5, $\mathcal{S}$ needs to simulate the behaviors of honest parties in EXTEND-COMPRESS. Note that EXTEND-COMPRESS only contains local computation and invocations of $\mathcal{F}_{\text{inner-product-mal}}, \mathcal{F}_{\text{coin}}$. For $\mathcal{F}_{\text{inner-product-mal}}$, the input sharings are linear combinations of $\{[\boldsymbol{a}^{(i,j)}], [\boldsymbol{b}^{(i,j)}]\}_{i\in[\kappa],j\in[\ell]}$, therefore $\mathcal{S}$ can computes the shares of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. $\mathcal{S}$ emulates $\mathcal{F}_{\text{inner-product-mal}}$ as above, and receives from the adversary the shares of the output sharing held by corrupted parties and the additive errors to the shares of the output sharing of parties in $\mathcal{H}_{\mathcal{C}}$. $\mathcal{S}$ also receives the additive errors to the secrets of the output sharing. For $\mathcal{F}_{\text{coin}}$, $\mathcal{S}$ faithfully generates a random element. Let

$$(([\boldsymbol{a}^{(1)}], \ldots, [\boldsymbol{a}^{(\ell)}]), ([\boldsymbol{b}^{(1)}], \ldots, [\boldsymbol{b}^{(\ell)}]), [\boldsymbol{c}])$$

denote the output of EXTEND-COMPRESS. By following the protocol, for each sharing, $\mathcal{S}$ can compute the shares of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Note that for each invocation of $\mathcal{F}_{\text{inner-product-mal}}$, the input sharings are linear combinations of $\{[\boldsymbol{a}^{(i,j)}], [\boldsymbol{b}^{(i,j)}]\}_{i\in[\kappa],j\in[\ell]}$, which are just $\{[\boldsymbol{x}^{(i)}], [\boldsymbol{y}^{(i)}]\}_{i=1}^{m}$. Therefore, if $\boldsymbol{\Delta}_x^{(i)} = \boldsymbol{\Delta}_y^{(i)} = \boldsymbol{0}$ for all $i \in [m]$, $\mathcal{S}$ learns the additive errors to the inner-product results of each inner-product tuple computed by $\mathcal{F}_{\text{inner-product-mal}}$. $\mathcal{S}$ further computes the additive errors $\boldsymbol{c} - \sum_{j=1}^{\ell} \boldsymbol{a}^{(\ell)} * \boldsymbol{b}^{(\ell)}$.

Note that for all $j \in [\ell]$, $[\boldsymbol{a}^{(j)}], [\boldsymbol{b}^{(j)}]$ are linear combinations of $\{[\boldsymbol{x}^{(i)}]\}_{i=1}^{m}, \{[\boldsymbol{y}^{(i)}]\}_{i=1}^{m}$ respectively. If $\boldsymbol{\Delta}_x^{(i)} = \boldsymbol{\Delta}_y^{(i)} = \boldsymbol{0}$ for all $i \in [m]$, then $[\boldsymbol{a}^{(j)}], [\boldsymbol{b}^{(j)}]$ are consistent for all $j \in [\ell]$. In this case, $\mathcal{S}$ also computes the additive errors $\boldsymbol{c} - \sum_{j=1}^{\ell} \boldsymbol{a}^{(\ell)} * \boldsymbol{b}^{(\ell)}$. This maintains the invariant for the next invocation of DIMENSION-REDUCTION. If any of the following conditions holds, $\mathcal{S}$ aborts:

- if for all $j \in [\ell]$, $[\boldsymbol{a}^{(j)}]$ is consistent, but there exists $i \in [m]$ such that $\boldsymbol{\Delta}_x^{(i)} \neq \boldsymbol{0}$;
- if for all $j \in [\ell]$, $[\boldsymbol{b}^{(j)}]$ is consistent, but there exists $i \in [m]$ such that $\boldsymbol{\Delta}_y^{(i)} \neq \boldsymbol{0}$;
- if $[\boldsymbol{c}]$ is consistent but $\boldsymbol{\Delta}_z \neq \boldsymbol{0}$;
- if the additive errors $\boldsymbol{c} - \sum_{j=1}^{\ell} \boldsymbol{a}^{(\ell)} * \boldsymbol{b}^{(\ell)} = \boldsymbol{0}$ but $\boldsymbol{d} = \boldsymbol{z} - \sum_{i=1}^{m} \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)} \neq \boldsymbol{0}$.

Note that these four conditions correspond to the four conditions in Lemma 22 respectively. According to Lemma 22, it only happens with negligible probability.

- Simulation of RANDOMIZATION:

  Note that, for the input inner-product tuple

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

$\mathcal{S}$ learns the shares held by corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ which are denoted by $\{\boldsymbol{\Delta}_x^{(i)}\}_{i=1}^{\kappa}, \{\boldsymbol{\Delta}_y^{(i)}\}_{i=1}^{\kappa}, \boldsymbol{\Delta}_z$ respectively. In addition, if $\boldsymbol{\Delta}_x^{(i)} = \boldsymbol{\Delta}_y^{(i)} = \mathbf{0}$ for all $i \in [\kappa]$, then $\mathcal{S}$ learns the additive errors to the inner-product results, i.e., $\boldsymbol{d} = \boldsymbol{z} - \sum_{i=1}^{\kappa} \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$. It follows from the invariant we maintain when simulating Dimension-Reduction.

In Step 2, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ when preparing $[\boldsymbol{x}^{(0)}], [\boldsymbol{y}^{(0)}]$. $\mathcal{S}$ receives from the adversary the shares of corrupted parties. Note that $\mathcal{F}_{\text{rand}}$ guarantees that $[\boldsymbol{x}^{(0)}], [\boldsymbol{y}^{(0)}]$ are consistent. Therefore, $\boldsymbol{\Delta}_x^{(0)} = \boldsymbol{\Delta}_y^{(0)} = \mathbf{0}$.

In Step 3, $\mathcal{S}$ emulates $\mathcal{F}_{\text{mult-mal}}$ when computing $[\boldsymbol{z}^{(0)}] = [\boldsymbol{x}^{(0)} * \boldsymbol{y}^{(0)}]$. $\mathcal{S}$ sends to the adversary the shares of $[\boldsymbol{x}^{(0)}], [\boldsymbol{y}^{(0)}]$ held by corrupted parties and $\boldsymbol{\Delta}_x^{(0)}, \boldsymbol{\Delta}_y^{(0)}$. Then $\mathcal{S}$ receives the shares of $[\boldsymbol{z}^{(0)}]$ of corrupted parties, the additive errors $\boldsymbol{\Delta}_z^{(0)}$ to the shares of $[\boldsymbol{z}^{(0)}]$ of parties in $\mathcal{H}_{\mathcal{C}}$, and the additive errors $\boldsymbol{d}^{(0)}$ to the secrets of $[\boldsymbol{z}^{(0)}]$. Since $\boldsymbol{\Delta}_x^{(0)} = \boldsymbol{\Delta}_y^{(0)} = \mathbf{0}$, $\boldsymbol{d}^{(0)} = \boldsymbol{z}^{(0)} - \boldsymbol{x}^{(0)} * \boldsymbol{y}^{(0)}$.

Step 4 and Step 5 (including Compress) can be simulated in the same way as that for Dimension-Reduction. Let $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$ denote the output of Compress in Step 5. For the output multiplication tuple, $\mathcal{S}$ learns the shares of corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ which are denoted by $\boldsymbol{\Delta}_a, \boldsymbol{\Delta}_b, \boldsymbol{\Delta}_c$ respectively. If $\boldsymbol{\Delta}_x^{(i)} = \boldsymbol{\Delta}_y^{(i)} = \mathbf{0}$ for all $i \in [\kappa]$, $\mathcal{S}$ also computes the additive errors $\boldsymbol{c} - \boldsymbol{a} * \boldsymbol{b}$. If any of the following conditions holds, $\mathcal{S}$ aborts:

- if $[\boldsymbol{a}]$ is consistent, but there exists $i \in [\kappa]$ such that $\boldsymbol{\Delta}_x^{(i)} \neq \mathbf{0}$;
- if $[\boldsymbol{b}]$ is consistent, but there exists $i \in [\kappa]$ such that $\boldsymbol{\Delta}_y^{(i)} \neq \mathbf{0}$;
- if $[\boldsymbol{c}]$ is consistent but $\boldsymbol{\Delta}_z \neq \mathbf{0}$;
- if the additive errors $\boldsymbol{c} - \boldsymbol{a} * \boldsymbol{b} = \mathbf{0}$ but $\boldsymbol{d} = \boldsymbol{z} - \sum_{i=1}^{\kappa} \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)} \neq \mathbf{0}$.

Note that $\boldsymbol{\Delta}_z = \sum_{i=1}^{\kappa} \boldsymbol{\Delta}_z^{(i)}$ and $\boldsymbol{d} = \sum_{i=1}^{\kappa} \boldsymbol{d}_z^{(i)}$. Therefore, if $\boldsymbol{\Delta}_z \neq \mathbf{0}$, then there exists $i \in [\kappa]$ such that $[\boldsymbol{z}^{(i)}]$ is inconsistent. If $\boldsymbol{d} \neq \mathbf{0}$, then there exists $i \in [\kappa]$ such that $\boldsymbol{d}^{(i)} = \boldsymbol{z}^{(i)} - \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)} \neq \mathbf{0}$. These four conditions correspond Condition 2 to Condition 5 in Lemma 18. By Lemma 18, it only happens with negligible probability.

In Step 6, $\mathcal{S}$ samples two random vectors as $\boldsymbol{a}, \boldsymbol{b}$. Then $\mathcal{S}$ reconstructs the whole sharings $[\boldsymbol{a}], [\boldsymbol{b}]$ using the secrets $\boldsymbol{a}, \boldsymbol{b}$, the shares of corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. $\mathcal{S}$ sends the shares of $[\boldsymbol{a}], [\boldsymbol{b}]$ of honest parties to corrupted parties and then honestly checks the consistency of $[\boldsymbol{a}], [\boldsymbol{b}]$.

In Step 7, $\mathcal{S}$ computes $\boldsymbol{c}$ using $\boldsymbol{a}, \boldsymbol{b}$ and the additive errors $\boldsymbol{c} - \boldsymbol{a} * \boldsymbol{b}$. $\mathcal{S}$ reconstructs the whole sharing $[\boldsymbol{c}]$ using the secrets $\boldsymbol{c}$, the shares of corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. $\mathcal{S}$ sends the shares of $[\boldsymbol{c}]$ of honest parties to corrupted parties and then honestly checks the correctness of the multiplication tuple. Recall that $\mathcal{S}$ receives $b \in \{\texttt{abort}, \texttt{accept}\}$ from $\mathcal{F}_{\text{mult-veri}}$ indicating whether the multiplications are correct. If $b = \texttt{accept}$ but an honest party aborts, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{mult-veri}}$. Otherwise, $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\text{mult-veri}}$.

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, for each sharing, $\mathcal{S}$ computes the shares of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ as described above. For each multiplication tuple or each inner-product tuple, $\mathcal{S}$ computes the additive errors to the multiplication results or the inner-product results as described above. Then, $\mathcal{S}$ checks the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ and the additive errors to the multiplication results and the inner-product results. Note that this does not change the behaviors of honest parties. By Lemma 21, Lemma 22 and Lemma 18, the probability that $\mathcal{S}$ aborts during the check of the additive errors is negligible. Therefore, the distribution of **Hybrid**$_1$ is statistically close to **Hybrid**$_0$.

$\mathbf{Hybrid}_2$: In this hybrid, instead of using the real sharings $[\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}]$ in Randomization, $\mathcal{S}$ constructs the sharings $[\boldsymbol{a}], [\boldsymbol{b}]_t, [\boldsymbol{c}]$ as described above. Concretely, $\mathcal{S}$ randomly samples the secrets $\boldsymbol{a}, \boldsymbol{c}$ and reconstructs the whole sharings $[\boldsymbol{a}], [\boldsymbol{b}]$ based on the secrets $\boldsymbol{a}, \boldsymbol{b}$, the shares of corrupted parties, and the additive errors to the shares of $[\boldsymbol{a}], [\boldsymbol{b}]$ of parties in $\mathcal{H}_{\mathcal{C}}$. If $\mathcal{S}$ does not abort when checking the additive errors in Randomization and $[\boldsymbol{a}], [\boldsymbol{b}]$ are consistent, $\mathcal{S}$ has computed the additive errors $\boldsymbol{c} - \boldsymbol{a} * \boldsymbol{b}$. $\mathcal{S}$ computes the secrets $\boldsymbol{c}$ using $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c} - \boldsymbol{a} * \boldsymbol{c}$ and reconstructs the whole sharing $[\boldsymbol{c}]$ based on the secrets $\boldsymbol{c}$, the shares of corrupted parties, and the additive errors to the shares of $[\boldsymbol{c}]$ of parties in $\mathcal{H}_{\mathcal{C}}$.

Note that in Randomization, $\boldsymbol{a}$ and $\boldsymbol{b}$ are linear combinations of $\{\boldsymbol{x}^{(i)}\}_{i=0}^{m}$ and $\{\boldsymbol{b}^{(i)}\}_{i=0}^{m}$ respectively and the coefficients are all non-zero, where the latter follows from the property of polynomials. Also note that $\boldsymbol{x}^{(0)}$ and $\boldsymbol{y}^{(0)}$ are randomly chosen by $\mathcal{F}_{\text{rand}}$. Thus, $\boldsymbol{a}$ and $\boldsymbol{b}$ are uniformly random. The only difference between $\mathbf{Hybrid}_1$ and $\mathbf{Hybrid}_2$ is that, in $\mathbf{Hybrid}_1$, $\boldsymbol{a}$ and $\boldsymbol{b}$ are masked by $\boldsymbol{x}^{(0)}$ and $\boldsymbol{y}^{(0)}$ which are randomly chosen by $\mathcal{F}_{\text{rand}}$, while in $\mathbf{Hybrid}_2$, $\boldsymbol{a}$ and $\boldsymbol{b}$ are randomly chosen by $\mathcal{S}$. However the distribution of $\boldsymbol{a}$ and $\boldsymbol{b}$ remains unchanged. Since $\boldsymbol{c}$ is determined by $\boldsymbol{a}, \boldsymbol{b}$ and the additive errors $\boldsymbol{c} - \boldsymbol{a} * \boldsymbol{b}$, the distributions of $\boldsymbol{c}$ in both hybrids are the same. For each sharing of $[\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}]$, the whole sharing is determined by the secrets, the shares of corrupted parties, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Therefore, the distributions of $([\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}])$ in both hybrids are identical.

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

$\mathbf{Hybrid}_3$: In this hybrid, $\mathcal{S}$ simulates the behaviors of honest parties as described above. Note that honest parties need to communicate with corrupted parties only in Step 6 and Step 7 in Randomization where all parties verify the correctness of the final multiplication tuple. However, the preparation of $[\boldsymbol{a}], [\boldsymbol{b}], [\boldsymbol{c}]$ can be prepared by $\mathcal{S}$ without knowing the shares of honest parties. Also, when $\mathcal{S}$ emulating $\mathcal{F}_{\text{mult-mal}}$ and $\mathcal{F}_{\text{inner-product-mal}}$, the shares of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ that $\mathcal{S}$ needs to send to the adversary have been computed since $\mathbf{Hybrid}_1$. Therefore, emulating $\mathcal{F}_{\text{mult-mal}}$ and $\mathcal{F}_{\text{inner-product-mal}}$ does not need the shares of honest parties.

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

Note that $\mathbf{Hybrid}_3$ is the execution in the ideal world, and the distribution of $\mathbf{Hybrid}_3$ is statistically close to the distribution of $\mathbf{Hybrid}_0$, the execution in the real world. □

Concrete Efficiency. Now we analyze the communication complexity of Mult-veri. Recall that each time of running Dimension-Reduction reduces the dimension of the inner-product tuple by a factor of κ. Therefore, Mult-veri includes 1 invocation of De-Linearization, $(\log_\kappa m - 1)$ invocations of Dimension-Reduction and 1 invocation of Randomization. Since κ is the security parameter, $\log_\kappa m$ is bounded by a constant. The communication complexity of Mult-veri is $O(n^3 \cdot \kappa^2)$ bits.

7.2 Network Routing

In this subsection, we discuss how to verify the network routing. Recall that the network routing including

- evaluating the fan-out gates we insert (see Section 4.4),
- performing permutations on the secrets of the output sharings of each layer to achieve the non-collision property (see Section 4.2 for performing permutation, and Section 4.3 for the non-collision property),
- selecting secrets from the output sharings in previous layers to form input sharings for the current layer (see Section 4.2 for the idea of $\mathcal{F}_{\text{select-semi}}$),
- and finally permuting the secrets of each input sharing to the correct order (see Section 4.2).

Instead of verifying every invocation of every protocol, we choose to focus on the sharings before the network routing and the sharings after the network routing. Note that the network routing does not change the secret values. Instead, its goal is to create new sharings which contain the secret values we want in the correct positions.

For each input sharing $[\boldsymbol{y}]$ of multiplication gates, addition gates, and output gates, and for each position $j \in [k]$, suppose y_j comes from the i-th secret of $[\boldsymbol{x}]$ which is an output sharing of input gates, multiplication gates, or addition gates in a previous layer. To verify the correctness of y_j, it is sufficient to check whether $x_i = y_j$. Note that this corresponds to the wire which carries the value $x_i = y_j$ in the circuit. In general, let m denote the number of wires in the circuit. For all $\ell \in [m]$, suppose the value carried by the ℓ-th wire is stored in the i_ℓ-th secret of $[\boldsymbol{x}^{(\ell)}]$ and the j_ℓ-th secret of $[\boldsymbol{y}^{(\ell)}]$. We want to verify that $x_{i_\ell}^{(\ell)} = y_{j_\ell}^{(\ell)}$. In other words, given m tuples

$$([\boldsymbol{x}^{(1)}], [\boldsymbol{y}^{(1)}], i_1, j_1), ([\boldsymbol{x}^{(2)}], [\boldsymbol{y}^{(2)}], i_2, j_2), \ldots, ([\boldsymbol{x}^{(m)}], [\boldsymbol{y}^{(m)}], i_m, j_m),$$

our goal is to check that $\forall \ell \in [m]$, $x_{i_\ell}^{(\ell)} = y_{j_\ell}^{(\ell)}$. We refer to a tuple $([\boldsymbol{x}], [\boldsymbol{y}], i, j)$ as a wire tuple. The functionality $\mathcal{F}_{\text{network}}$ appears in Functionality 44. The functionality $\mathcal{F}_{\text{network}}$ receives the wire tuples from honest parties. For each wire tuple $([\boldsymbol{x}], [\boldsymbol{y}], i, j)$, $\mathcal{F}_{\text{network}}$ checks whether $x_i = y_j$. In addition, $\mathcal{F}_{\text{network}}$ sends the difference $y_j - x_i$ to the adversary. Note that, for protocols we need to run in the network routing, the attacks that an adversary can do are limited to

- Adding additive errors to the secrets of the output sharings of these protocols,
- Adding additive errors to the shares of the output sharings of parties in $\mathcal{H}_\mathcal{C}$.

Note that the additive errors to the shares of parties in $\mathcal{H}_\mathcal{C}$ do not affect the secrets since they are determined by the shares of parties in $\mathcal{H}_\mathcal{H}$. Therefore, the adversary knows the additive values to the wire w along the way from x_i to y_j, which means that we can safely reveal $y_j - x_i$ to the adversary.

Functionality 44: $\mathcal{F}_{\text{network}}$

1. Let m denote the number of wire tuples all parties want to verify. For all $\ell \in [m]$, $\mathcal{F}_{\text{network}}$ receives from honest parties their shares of $[\boldsymbol{x}^{(\ell)}], [\boldsymbol{y}^{(\ell)}]$ and two positions $i_\ell, j_\ell \in [k]$. The ℓ-th tuple is stored $([\boldsymbol{x}^{(\ell)}], [\boldsymbol{y}^{(\ell)}], i_\ell, j_\ell)$.
2. For all $\ell \in [m]$, $\mathcal{F}_{\text{network}}$ reconstructs the whole sharings $[\boldsymbol{x}^{(\ell)}]_\mathcal{H}, [\boldsymbol{x}^{(\ell)}], [\boldsymbol{y}^{(\ell)}]_\mathcal{H}, [\boldsymbol{y}^{(\ell)}]$ using the shares of honest parties and computes $\boldsymbol{\Delta}_x^{(\ell)} := [\boldsymbol{x}^{(\ell)}] - [\boldsymbol{x}^{(\ell)}]_\mathcal{H}$ and $\boldsymbol{\Delta}_y^{(\ell)} := [\boldsymbol{y}^{(\ell)}] - [\boldsymbol{y}^{(\ell)}]_\mathcal{H}$. Then $\mathcal{F}_{\text{network}}$ sends to the adversary the shares of $[\boldsymbol{x}^{(\ell)}]_\mathcal{H}, [\boldsymbol{y}^{(\ell)}]_\mathcal{H}$ of corrupted parties and the vectors $\boldsymbol{\Delta}_x^{(\ell)}, \boldsymbol{\Delta}_y^{(\ell)}$.
3. For all $\ell \in [m]$, $\mathcal{F}_{\text{network}}$ reconstructs $\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}$ and computes $d^{(\ell)} = y_{j_\ell}^{(\ell)} - x_{i_\ell}^{(\ell)}$. Then $\mathcal{F}_{\text{network}}$ sends $d^{(\ell)}$ to the adversary.
4. Let $b \in \{\texttt{abort}, \texttt{accept}\}$ denote whether there exists $\ell \in [m]$ such that $d^{(\ell)} \neq 0$. $\mathcal{F}_{\text{network}}$ sends b to the adversary and waits for its response.
   - If the adversary replies `continue`, $\mathcal{F}_{\text{network}}$ sends b to honest parties.
   - If the adversary replies `abort`, $\mathcal{F}_{\text{network}}$ sends `abort` to honest parties.

In the following, we will lift all the sharings into a large enough extension field of $\mathbb{F}$, denoted by $\mathbb{K}$, such that $|\mathbb{K}| \geq 2^\kappa$. We assume that κ is the size of a field element in $\mathbb{K}$. To realize $\mathcal{F}_{\text{network}}$, the idea is to first classify all the wire tuples based on the positions we want to verify. Specifically, for all $i, j \in \{1, 2, \ldots, k\}$, all parties construct a list $L(i, j)$ containing all the wire tuples whose last two entries are i, j. To check the correctness of the wire tuples in $L(i, j)$, all parties compute a random linear combination of the tuples in $L(i, j)$ and obtain two sharings $([\boldsymbol{x}^\star], [\boldsymbol{y}^\star])$. Note that if all the wire tuples are correct, we should have $x_i^\star = y_j^\star$. On the other hand, we will show that with overwhelming probability, if there exists an incorrect wire tuple in $L(i, j)$, then $\boldsymbol{x}_i^\star \neq \boldsymbol{y}_j^\star$. To simplify the verification of the wire tuple $([\boldsymbol{x}^\star], [\boldsymbol{y}^\star], i, j)$, all parties will use $\mathcal{F}_{\text{rand}}$ to prepare a random wire tuple $([\boldsymbol{x}^{(0)}], [\boldsymbol{y}^{(0)}], i, j)$ and add it into the list $L(i, j)$. This random wire tuple serves as a random mask to $([\boldsymbol{x}^\star],[\boldsymbol{y}^\star],i,j)$ so that the verification of $([\boldsymbol{x}^\star],[\boldsymbol{y}^\star],i,j)$ can be done by simply opening $[\boldsymbol{x}^\star],[\boldsymbol{y}^\star]$.

For fixed i,j, We show how to use $\mathcal{F}_{\text{rand}}$ to prepare a random wire tuple $([\boldsymbol{x}],[\boldsymbol{y}],i,j)$ in Appendix B.5. The communication complexity of preparing m random wire tuples for fixed i,j is $O(m\cdot n+n^3\cdot\kappa)$ elements in $\mathbb{K}$. In the implementation of $\mathcal{F}_{\text{network}}$, we only need one random wire tuple for each $(i,j)\in\{1,2,\ldots,k\}^2$. Therefore, the communication complexity of preparing the random wire tuples is $O(n^5\cdot\kappa)$ elements in $\mathbb{K}$, which is $O(n^5\cdot\kappa^2)$ bits. The description of Network appears in Protocol 45. The communication complexity of Network is $O(n^5\cdot\kappa^2)$ bits.

Protocol 45: Network

1. Let m denote the number of wire tuples all parties want to verify. The wire tuples are denoted by
$$([\boldsymbol{x}^{(1)}],[\boldsymbol{y}^{(1)}],i_1,j_1),([\boldsymbol{x}^{(2)}],[\boldsymbol{y}^{(2)}],i_2,j_2),\ldots,([\boldsymbol{x}^{(m)}].[\boldsymbol{y}^{(m)}],i_m,j_m).$$
2. For all $i,j\in\{1,2,\ldots,k\}$, all parties initiate an empty list $L(i,j)$. From $\ell=1$ to m, all parties insert $([\boldsymbol{x}^{(\ell)}],[\boldsymbol{y}^{(\ell)}],i_\ell,j_\ell)$ into the list $L(i_\ell,j_\ell)$.
3. For all $i,j\in\{1,2,\ldots,k\}$, all parties verify the wire tuples in $L(i,j)$ as follows:
   (a) Let m' denote the size of $L(i,j)$ and the wire tuples in $L(i,j)$ are denoted by
   $$([\boldsymbol{a}^{(1)}],[\boldsymbol{b}^{(1)}],i,j),([\boldsymbol{a}^{(2)}],[\boldsymbol{b}^{(2)}],i,j),\ldots,([\boldsymbol{a}^{(m')}].[\boldsymbol{b}^{(m')}],i,j).$$
   (b) All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare a random wire tuple $([\boldsymbol{a}^{(0)}],[\boldsymbol{b}^{(0)}],i,j)$ in $\mathbb{K}$.
   (c) All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random element $r\in\mathbb{K}$.
   (d) All parties compute the following two sharings:
   $$\begin{aligned}[\boldsymbol{a}] &= \sum_{\ell=0}^{m'} r^i\cdot[\boldsymbol{a}^{(\ell)}]\\ [\boldsymbol{b}] &= \sum_{\ell=0}^{m'} r^i\cdot[\boldsymbol{b}^{(\ell)}]\end{aligned}$$
   (e) All parties send their shares of $[\boldsymbol{a}],[\boldsymbol{b}]$ to other parties. Then each party checks whether $[\boldsymbol{a}],[\boldsymbol{b}]$ are consistent and $a_i=b_j$. If not , this party aborts.

Lemma 25. *Protocol* Network *(see Protocol 45) securely computes $\mathcal{F}_{network}$ in the $(\mathcal{F}_{rand},\mathcal{F}_{coin})$-hybrid model against a fully malicious adversary who controls $t'=t-k+1$ parties with overwhelming probability.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that $\mathcal{H}_{\mathcal{H}}\subset\mathcal{H}$ is a fixed subset of size $t+1$.

Simulation for Network. In the beginning, for all $\ell\in[m]$ $\mathcal{S}$ receives from $\mathcal{F}_{\text{network}}$ the shares of $[\boldsymbol{x}^{(\ell)}]_{\mathcal{H}},[\boldsymbol{y}^{(\ell)}]_{\mathcal{H}}$ held by corrupted parties, the additive errors $\boldsymbol{\Delta}_x^{(\ell)},\boldsymbol{\Delta}_y^{(\ell)}$ to the shares of $[\boldsymbol{x}^{(\ell)}],[\boldsymbol{y}^{(\ell)}]$ of parties in $\mathcal{H}_{\mathcal{C}}$, and the additive error $d^{(\ell)}=y_{j_\ell}^{(\ell)}-x_{i_\ell}^{(\ell)}$.

- In the first two steps, $\mathcal{S}$ honestly follow the protocol to classify the wire tuples based on the positions.
- In Step 3.(a), let
$$([\boldsymbol{a}^{(1)}],[\boldsymbol{b}^{(1)}],i,j),([\boldsymbol{a}^{(2)}],[\boldsymbol{b}^{(2)}],i,j),\ldots,([\boldsymbol{a}^{(m')}].[\boldsymbol{b}^{(m')}],i,j).$$
denote the wire tuples in $L(i,j)$. For all $\ell \in [m']$, $\mathcal{S}$ learns the shares of $[\boldsymbol{a}^{(\ell)}]_{\mathcal{H}}, [\boldsymbol{b}^{(\ell)}]_{\mathcal{H}}$ held by corrutped parties, the additive errors $\boldsymbol{\Delta}_a^{(\ell)}, \boldsymbol{\Delta}_b^{(\ell)}$ to the shares of $[\boldsymbol{a}^{(\ell)}], [\boldsymbol{b}^{(\ell)}]$ of parties in $\mathcal{H}_{\mathcal{C}}$, and the additive error $\tilde{d}^{(\ell)} = b_j^{(\ell)} - a_i^{(\ell)}$.
- In Step 3.(b), $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ when preparing a random wire tuple $([\boldsymbol{a}^{(0)}],[\boldsymbol{b}^{(0)}],i,j)$. $\mathcal{S}$ receives the shares of $[\boldsymbol{a}^{(0)}],[\boldsymbol{b}^{(0)}]$ held by corrupted parties. Note that $[\boldsymbol{a}^{(0)}],[\boldsymbol{b}^{(0)}]$ are guaranteed to be consistent, and $a_i^{(0)} = b_j^{(0)}$. Therefore, $\boldsymbol{\Delta}_a^{(0)} = \boldsymbol{\Delta}_b^{(0)} = \boldsymbol{0}$ and $\tilde{d}^{(0)} = 0$.
- In Step 3.(c), $\mathcal{S}$ emulates $\mathcal{F}_{\text{coin}}$ and faithfully samples $r \in \mathbb{K}$.
- In Step 3.(d), $\mathcal{S}$ computes $\boldsymbol{\Delta}_a = \sum_{i=0}^{m'} r^i \cdot \boldsymbol{\Delta}_a^{(i)}$, $\boldsymbol{\Delta}_b = \sum_{i=0}^{m'} r^i \cdot \boldsymbol{\Delta}_b^{(i)}$, and $\tilde{h} = \sum_{i=0}^{m'} r^i \cdot \tilde{h}^{(i)}$. $\mathcal{S}$ also computes the shares of $[\boldsymbol{a}],[\boldsymbol{b}]$ held by corrupted parties. Then, $\mathcal{S}$ samples a random vector in $\mathbb{K}^k$ as $\boldsymbol{a}$. Based on the secrets $\boldsymbol{a}$, the shares of $[\boldsymbol{a}]$ held by corrupted parties, and the additive errors $\boldsymbol{\Delta}_a$ to the shares of parties in $\mathcal{H}_{\mathcal{C}}$, $\mathcal{S}$ reconstructs the whole sharing $[\boldsymbol{a}]$. For $[\boldsymbol{b}]$, $\mathcal{S}$ sets $b_j = a_i + \tilde{d}$ and samples $k-1$ random elements in $\mathbb{K}$ as the values for $\{b_\ell\}_{\ell \neq j}$. Based on the secrets $\boldsymbol{b}$, the shares of $[\boldsymbol{b}]$ held by corrupted parties, and the additive errors $\boldsymbol{\Delta}_b$ to the shares of parties in $\mathcal{H}_{\mathcal{C}}$, $\mathcal{S}$ reconstructs the whole sharing $[\boldsymbol{b}]$.
- In Step 3.(e), $\mathcal{S}$ honestly follows the protocol.
- Finally, if an honest party aborts, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{network}}$. Otherwise, $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\text{network}}$.

Analyze the Security of Network. We first show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Note that honest parties need to send messages to corrupted parties only in step 3.(e). Relying on the values received from $\mathcal{F}_{\text{network}}$ in the beginning and the values received from the adversary when emulating $\mathcal{F}_{\text{rand}}$, $\mathcal{S}$ can compute the shares of $[\boldsymbol{a}],[\boldsymbol{b}]$ held by corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Furthermore, $\mathcal{S}$ can compute the additive error $\tilde{d} = b_j - a_i$. To determine the whole sharings $[\boldsymbol{a}],[\boldsymbol{b}]$, $\mathcal{S}$ only needs to know (the distribution) of $\boldsymbol{a},\boldsymbol{b}$. In the real world, $\boldsymbol{a},\boldsymbol{b}$ are masked by $\boldsymbol{a}^{(0)},\boldsymbol{b}^{(0)}$, which are random subject to $a_i^{(0)} = b_j^{(0)}$ guaranteed by $\mathcal{F}_{\text{rand}}$. Therefore, $\boldsymbol{a}$ is a uniform vector, $\{b_\ell\}_{\ell \neq j}$ are uniform elements in $\mathbb{K}$, and $b_j = a_i + \tilde{d}$. In the ideal world, $\mathcal{S}$ randomly samples $\boldsymbol{a}$ and $\{b_\ell\}_{\ell \neq j}$, and sets $b_j = a_i + \tilde{d}$, which have the same distribution as that in the real world. Therefore, $\mathcal{S}$ perfectly simulates the behaviors of honest parties.

Now we analyze the distribution of the output. The only difference is when there exists a wire tuple $([\boldsymbol{a}^{(\ell)}],[\boldsymbol{b}^{(\ell)}],i,j)$ such that $a_i^\ell \neq b_j^\ell$ but for the wire tuple $([\boldsymbol{a}],[\boldsymbol{b}],i,j)$, $a_i = b_j$. In the ideal world, all parties will finally abort, while in the real world, all parties will continue. We show that this happens with negligible probability. Note that it is sufficient to show that, if there exists $\ell \in [m']$ such that $\tilde{d}^{(\ell)} \neq 0$, then with overwhelming probability, $\tilde{d} \neq 0$.

Consider the polynomial $\delta(\cdot) = \tilde{d}^{(0)} + \tilde{d}^{(1)} \cdot r + \tilde{d}^{(m')} \cdot r^{m'}$. Then we have $\tilde{d} = \delta(r)$. If there exists $\ell \in [m']$ such that $\tilde{d}^{(\ell)} \neq 0$, then $\delta(\cdot)$ is a non-zero polynomial. Since $\delta(\cdot)$ is of degree-m', the number of r such that $\delta(r) = 0$ is bounded by m'. Since r is randomly sampled by $\mathcal{F}_{\text{coin}}$ in $\mathbb{K}$, the probability that $\delta(r) = 0$ is bounded by $m'/|\mathbb{K}| \leq m'/2^\kappa$, which is negligible. Therefore, with overwhelming probability, $\tilde{d} \neq 0$. □

8 Main Protocol - Against a Fully Malicious Adversary

In this section, we will introduce our main protocol of using packed Shamir sharing to evaluate a general circuit C against a *fully malicious* adversary. As we discussed in Section 6, most of our semi-honest protocols constructed in Section 4 achieve perfect privacy against a fully malicious adversary. Therefore, to achieve malicious security, our idea is to run our semi-honest protocol described in Section 5 before the output phase, check the correctness of the computation, and finally reconstruct the output only if the verification passes. As we discussed in Section 7, we view the computation as a composition of two parts: (1) evaluation of the basic gates, i.e., addition gates and multiplication gates, and (2) network routing, i.e., computing input sharings of each layer using the output sharings of previous layers. All parties will use the two functionalities we introduced in Section 7 to check these two parts.

Recall that we are in the client-server model where there are c clients and $n = 2t + 1$ parties (servers). Recall that $1 \leq k \leq t$ is an integer. An adversary is allowed to corrupt $t' = t - k + 1$ parties. We will use the degree-t packed Shamir sharing scheme, which can store k secrets within one sharing. Recall that $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties.

The ideal functionality $\mathcal{F}_{\text{main-mal}}$ appears in Functionality 46.

Functionality 46: $\mathcal{F}_{\text{main-mal}}$

1. $\mathcal{F}_{\text{main-mal}}$ receives the input from all clients. Let x denote the input.
2. $\mathcal{F}_{\text{main-mal}}$ computes $C(x)$ and sends the output of corrupted clients to the adversary. $\mathcal{F}_{\text{main-mal}}$ waits for the response of the adversary.
    - If the adversary replies `abort`, $\mathcal{F}_{\text{main-mal}}$ sends `abort` to all clients.
    - If the adversary replies `continue`, $\mathcal{F}_{\text{main-mal}}$ distributes the output to all clients.

We will separate the main protocol into two parts. The first part follows the semi-honest protocol till Step 4.(a), where all parties prepare the input sharings of the output layers, with the modification that the invocations of semi-honest functionalities are replaced by the malicious variants. The second part includes the verification of multiplications and network routing and Step 4.(b) of the semi-honest protocol, where all parties reconstruct the output. The first part appears in Protocol 47 and the second part appears in Protocol 48.

Lemma 26. *Protocol* MAIN *(see Protocol 47 and Protocol 48) securely computes $\mathcal{F}_{main\text{-}mal}$ in the $(\mathcal{F}_{input\text{-}mal}, \mathcal{F}_{fan\text{-}out\text{-}mal}, \mathcal{F}_{permute\text{-}mal}, \mathcal{F}_{select\text{-}mal}, \mathcal{F}_{mult\text{-}mal}, \mathcal{F}_{mult\text{-}veri}, \mathcal{F}_{network}, \mathcal{F}_{output\text{-}mal})$-hybrid model against a fully malicious adversary who controls $t' = t - k + 1$ parties.*

Proof. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Let $\mathcal{C}$ denote the set of corrupted parties and $\mathcal{H}$ denote the set of honest parties. Recall that $\mathcal{H}_{\mathcal{H}} \subset \mathcal{H}$ is a fixed subset of size $t + 1$.

Compared with the semi-honest protocol MAIN-SEMI in Section 5, the only difference is that all parties verify the correctness before reconstructing the output. Therefore, the correctness of MAIN follows from the same argument as that in Lemma 9.

Simulation of Main. We describe the strategy of $\mathcal{S}$ phase by phase. In the first step, $\mathcal{S}$ honestly follows the protocol PREPROCESS.

- Simulation of Input Phase:

  In Step 2.(a), $\mathcal{S}$ emulates $\mathcal{F}_{\text{input-mal}}$. For each corrupted Client_i, $\mathcal{S}$ receives the input of Client_i. For each output sharing, $\mathcal{S}$ receives from the adversary the shares of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Then, $\mathcal{S}$ sends the input of corrupted clients to $\mathcal{F}_{\text{main-mal}}$.

  In Step 2.(b), $\mathcal{S}$ emulates $\mathcal{F}_{\text{fan-out-mal}}$. For the input sharing $[\boldsymbol{x}]$, $\mathcal{S}$ first sends to the adversary the shares of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Note that they are

Protocol 47: Main-Part I

1. **Circuit Transformation Phase**. Let C denote the evaluated circuit. All parties preprocess the circuit by running the Preprocess protocol. Let C' denote the circuit after transformation.

2. **Input Phase**. Let $\mathsf{Client}_1, \mathsf{Client}_2, \ldots, \mathsf{Client}_c$ denote the clients who provide inputs, and Client_0 denote the virtual client who provides constants.

   (a) Input Secret-sharing Phase: For every group of k input gates of Client_i ($i \geq 1$), Client_i invoke $\mathcal{F}_{\text{input-mal}}$ to share its inputs $\boldsymbol{x}^{(i)}$ to the parties. For every group of k input gates of Client_0, the inputs $\boldsymbol{x}^{(0)}$ are constant values and known to all parties. All parties transform $\boldsymbol{x}^{(0)}$ to a degree-t packed Shamir sharing $[\boldsymbol{x}^{(0)}]$ by following the approach in Section 3.2.

   (b) Handling Fan-out Gates: For the output sharing $[\boldsymbol{x}]$ of each group of input gates, let n_i denote the number of times that the i-th secret of $\boldsymbol{x}$ is used in later layers. All parties invoke $\mathcal{F}_{\text{fan-out-mal}}$ with input $[\boldsymbol{x}]$ and $(n_1, n_2, \ldots, n_k)$.

   (c) Achieving Non-Collision Property for the Next Layers: For each output sharing $[\boldsymbol{y}]$ of the input layer, let p denote the permutation associated with it. All parties invoke $\mathcal{F}_{\text{permute-mal}}$ with input $[\boldsymbol{y}]$ and p.

3. **Evaluation Phase**. All parties evaluate the circuit layer by layer as follows:

   (a) Permute Input Sharings from Previous Layers: For each input sharing $[\boldsymbol{x}]$, let $[\boldsymbol{x}^{(i)}]$ denote the output sharing from previous layers which contains the i-th secret x_i, and let q_i denote the position of x_i in $[\boldsymbol{x}^{(i)}]$. According to the non-collision property, $q_1, q_2, \ldots, q_k$ is a permutation of $(1, 2, \ldots, k)$. Let $q(\cdot)$ be a permutation over $\{1, 2, \ldots, k\}$ such that $q(i) = q_i$. All parties invoke $\mathcal{F}_{\text{select-mal}}$ on $[\boldsymbol{x}^{(1)}], [\boldsymbol{x}^{(2)}], \ldots, [\boldsymbol{x}^{(k)}]$ and the permutation q. Let $[\boldsymbol{x}']$ denote the output of $\mathcal{F}_{\text{select-mal}}$. Then, all parties invoke $\mathcal{F}_{\text{permute-mal}}$ with input $[\boldsymbol{x}']$ and q to obtain $[\boldsymbol{x}]$.

   (b) Evaluating Multiplication Gates and Addition Gates: For each group of multiplication gates with input sharings $[\boldsymbol{x}], [\boldsymbol{y}]$, all parties invoke $\mathcal{F}_{\text{mult-mal}}$ with input $[\boldsymbol{x}], [\boldsymbol{y}]$. For each group of addition gates with input sharings $[\boldsymbol{x}], [\boldsymbol{y}]$, all parties locally compute $[\boldsymbol{x} + \boldsymbol{y}] = [\boldsymbol{x}] + [\boldsymbol{y}]$.

   (c) Handling Fan-out Gates: For the output sharing $[\boldsymbol{x}]$ of each group of multiplication gates or addition gates, all parties follow the same step as Step 2.(b) to handle fan-out gates.

   (d) Achieving Non-Collision Property: Follow Step 2.(c).

4. **Preparing Input sharings of the output layer**. For each input sharing $[\boldsymbol{x}]$, all parties follow the same step as Step 3.(a) to prepare $[\boldsymbol{x}]$.

learnt when $\mathcal{S}$ emulates $\mathcal{F}_{\text{input-mal}}$. For each output sharing, $\mathcal{S}$ receives from the adversary the shares of corrupted parties, the additive errors to the secrets of the output sharing, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$.

In Step 2.(c), $\mathcal{S}$ emulates $\mathcal{F}_{\text{permute-mal}}$. For the input sharing $[\boldsymbol{y}]$, $\mathcal{S}$ first sends to the adversary the shares of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Note that they are learnt when $\mathcal{S}$ emulates $\mathcal{F}_{\text{fan-out-mal}}$. For the output sharing, $\mathcal{S}$ receives from the adversary the shares of corrupted parties, the additive errors to the secrets of the output sharing, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$.

In the following, we maintain the invariant that for each degree-t packed Shamir sharing, $\mathcal{S}$ learns the shares of corrupted parties and the additive errors to the parties in $\mathcal{H}_{\mathcal{C}}$. Note that the invariant holds during the Input Phase.

Protocol 48: MAIN-PART II

5. **Verification Phase**.

   (a) Let m denote the number of multiplication tuples computed by $\mathcal{F}_{\text{mult-mal}}$. The multiplication tuples are denoted by

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

   All parties invoke $\mathcal{F}_{\text{mult-veri}}$ to check the correctness of the multiplication tuples.

   (b) All parties locally prepare the wire tuples as follows: For each input sharing $[\boldsymbol{y}]$ of multiplication gates, addition gates, and output gates, and for each position $j \in [k]$, suppose y_j comes from the i-th secret of $[\boldsymbol{x}]$ which is an output sharing of input gates, multiplication gates, or addition gates in a previous layer. All parties set a wire tuple to be $([\boldsymbol{x}], [\boldsymbol{y}], i, j)$. Let m' denote the number of wire tuples that all parties want to check. The wire tuples are denoted by

   $$([\boldsymbol{x}^{(1)}], [\boldsymbol{y}^{(1)}], i_1, j_1), ([\boldsymbol{x}^{(2)}], [\boldsymbol{y}^{(2)}], i_2, j_2), \ldots, ([\boldsymbol{x}^{(m')}], [\boldsymbol{y}^{(m')}], i_{m'}, j_{m'}).$$

   All parties invoke $\mathcal{F}_{\text{network}}$ to check the correctness of the network routing.

6. **Output Reconstruction**. For each group of output gates belonging to Client_i ($i \geq 1$), let $[\boldsymbol{x}]$ denote the input sharing. All parties invoke $\mathcal{F}_{\text{output-mal}}$ with input $[\boldsymbol{x}]$ to let Client_i learn the result $\boldsymbol{x}$.

- Simulation of Evaluation Phase:

  In Step 3.(a), $\mathcal{S}$ emulates $\mathcal{F}_{\text{select-semi}}$ and $\mathcal{F}_{\text{permute-mal}}$. For $\mathcal{F}_{\text{select-semi}}$, for each input sharing of $\mathcal{F}_{\text{select-semi}}$, $\mathcal{S}$ first sends to the adversary the shares of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. By the invariant, $\mathcal{S}$ learnt these values when simulating previous steps. For the output sharing, $\mathcal{S}$ receives from the adversary the shares of corrupted parties, the additive errors to the secrets of the output sharing, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. $\mathcal{S}$ emulates $\mathcal{F}_{\text{permute-mal}}$ in the same way as that in Step 2.(c).

  In Step 3.(b), for each group of multiplication gates, $\mathcal{S}$ emulates $\mathcal{F}_{\text{mult-mal}}$. For the input sharings $[\boldsymbol{x}], [\boldsymbol{y}]$, $\mathcal{S}$ first sends to the adversary the shares of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. By the invariant, $\mathcal{S}$ learnt these values when simulating previous steps. For the output sharing $[\boldsymbol{z}]$, $\mathcal{S}$ receives from the adversary the shares of corrpupted parties, the additive errors to the secrets of the output sharing, and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Let $\boldsymbol{d}$ denote the additive errors to the secrets of the output sharing. Note that $\boldsymbol{d} = \boldsymbol{z} - \boldsymbol{x} * \boldsymbol{y}$ holds if $[\boldsymbol{x}], [\boldsymbol{y}]$ are consistent, i.e., the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$ are $\boldsymbol{0}$. For each group of addition gates, $\mathcal{S}$ computes the shares of the output sharing of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$.

  $\mathcal{S}$ simulates Step 3.(c) and Step 3.(d) in the same way as that for Step 2.(b) and Step 2.(c). Note that the that for each degree-t packed Shamir sharing in Step 3, $\mathcal{S}$ learns the shares of corrupted parties and the additive errors to the parties in $\mathcal{H}_{\mathcal{C}}$. Therefore, the invariant holds.

- Simulation of Step 4: $\mathcal{S}$ simulates Step 4 in the same way as that for Step 3.(a). Note that the that for each degree-t packed Shamir sharing in Step 4, $\mathcal{S}$ learns the shares of corrupted parties and the additive errors to the parties in $\mathcal{H}_{\mathcal{C}}$. Therefore, the invariant holds.

- Simulation of Verification Phase:

In Step 5.(a), $\mathcal{S}$ emulates $\mathcal{F}_{\text{mult-veri}}$. For each multiplication tuple $([\boldsymbol{x}^{(i)}], [\boldsymbol{y}^{(i)}], [\boldsymbol{z}^{(i)}])$, $\mathcal{S}$ sends to the adversary the shares of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. They are learnt when $\mathcal{S}$ emulates $\mathcal{F}_{\text{mult-mal}}$ to compute this tuple. Note that $\mathcal{S}$ also receives the additive errors $\boldsymbol{d}^{(i)}$ to the secrets of the output sharing when emulates $\mathcal{F}_{\text{mult-mal}}$. Recall that if the additive errors to the shares of $([\boldsymbol{x}^{(i)}], [\boldsymbol{y}^{(i)}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ are $\mathbf{0}$, $\boldsymbol{d}^{(i)} = \boldsymbol{z}^{(i)} - \boldsymbol{x}^{(i)} * \boldsymbol{y}^{(i)}$. Therefore, if the additive errors to the shares of $([\boldsymbol{x}^{(i)}], [\boldsymbol{y}^{(i)}]$ of parties in $\mathcal{H}_{\mathcal{C}}$ are $\mathbf{0}$, $\mathcal{S}$ sends $\boldsymbol{d}^{(i)}$ to the adversary. $\mathcal{S}$ honestly follows the rest of steps in $\mathcal{F}_{\text{mult-veri}}$.

In Step 5.(b), $\mathcal{S}$ emulates $\mathcal{F}_{\text{network}}$. For each wire tuple $([\boldsymbol{x}], [\boldsymbol{y}], i, j)$, $\mathcal{S}$ first computes $y_j - x_i$. Note that to obtain y_j from x_i, all parties invoke $\mathcal{F}_{\text{fan-out-mal}}, \mathcal{F}_{\text{permute-mal}}$ in Step 2.(b) and Step 2.(c) (or Step 3.(c) and Step 3.(d) depending on whether x_i is an output of an input gate, an addition gate, or a multiplication gate), and then invoke $\mathcal{F}_{\text{select-mal}}, \mathcal{F}_{\text{permute-mal}}$ in Step 3.(a) (or Step 4.(a) depending on whether y_i is an input of an addition gate, a multiplication gate, or an output gate). In each of these invocations, $\mathcal{S}$ learns from the adversary the additive error to the value x_i. Therefore, $\mathcal{S}$ can compute $y_j - x_i$ by adding up all the additive error to the value x_i in these four invocations. Then, for each wire tuple $([\boldsymbol{x}], [\boldsymbol{y}], i, j)$, $\mathcal{S}$ sends to the adversary the shares of corrupted parties, the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$, and the additive error $y_j - x_i$. $\mathcal{S}$ honestly follows the rest of steps in $\mathcal{F}_{\text{network}}$.

- Simulation of Step 6: $\mathcal{S}$ emulates $\mathcal{F}_{\text{output-mal}}$. For the input sharing $[\boldsymbol{x}]$, $\mathcal{S}$ first sends to the adversary the shares of corrupted parties and the additive errors to the shares of parties in $\mathcal{H}_{\mathcal{C}}$. If the receiver is a corrupted client, $\mathcal{S}$ receives $\boldsymbol{x}$ from $\mathcal{F}_{\text{main-mal}}$ and sends $\boldsymbol{x}$ to the receiver. If the receiver is an honest client, $\mathcal{S}$ waits for the reply of the adversary. If the adversary replies `abort`, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{main-mal}}$. If for all invocations of $\mathcal{F}_{\text{output-mal}}$, the adversary replies `continue`, $\mathcal{S}$ sends `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.

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

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

$\mathbf{Hybrid}_2$: In this hybrid, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\text{output-mal}}$ with the modification that when the receiver is corrupted, $\mathcal{S}$ uses the output received from $\mathcal{F}_{\text{main-mal}}$. Note that the output phase is executed only when the computation is correct. The output received from $\mathcal{F}_{\text{main-mal}}$ is the same as the output computed in the protocol. Therefore, the distribution of $\mathbf{Hybrid}_2$ is identical to the distribution of $\mathbf{Hybrid}_1$.

$\mathbf{Hybrid}_3$: In this hybrid, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\text{mult-mal}}$ and receives the additive errors to the secrets of the output sharing. Then, in $\mathcal{F}_{\text{mult-veri}}$, $\mathcal{S}$ directly uses these additive errors for each multiplication tuple as above. Note that $\mathcal{F}_{\text{mult-veri}}$ only sends the additive errors to the adversary when the additive errors to the shares of the two input sharings of parties in $\mathcal{H}_{\mathcal{C}}$ are $\mathbf{0}$. As we argued above, in this case, the additive errors to the secrets of the output sharing are the same as the additive errors to the multiplication results. Therefore, the distribution of $\mathbf{Hybrid}_3$ is identical to the distribution of $\mathbf{Hybrid}_2$.

$\mathbf{Hybrid}_4$: In this hybrid, $\mathcal{S}$ honestly emulates $\mathcal{F}_{\text{fan-out-mal}}, \mathcal{F}_{\text{permute-mal}}, \mathcal{F}_{\text{select-mal}}$ and recieves the additive errors to the secrets of the output sharings. Then, in $\mathcal{F}_{\text{network}}$, $\mathcal{S}$ prepares the additive error of each wire tuple as above. The distribution of $\mathbf{Hybrid}_4$ is identical to the distribution of $\mathbf{Hybrid}_3$.

$\mathbf{Hybrid}_5$: In this hybrid, $\mathcal{S}$ computes the shares of corrupted parties and the additive errors of the shares of parties in $\mathcal{H}_{\mathcal{C}}$. Then $\mathcal{S}$ uses these values to emulates $\mathcal{F}_{\text{fan-out-mal}}, \mathcal{F}_{\text{permute-mal}}, \mathcal{F}_{\text{select-mal}}, \mathcal{F}_{\text{mult-mal}}, \mathcal{F}_{\text{mult-veri}}, \mathcal{F}_{\text{network}}, \mathcal{F}_{\text{output-mal}}$ as above. The only difference is that in $\mathbf{Hybrid}_4$, these functionalities use the real shares of corrupted parties and the additive errors of the shares of parties in $\mathcal{H}_{\mathcal{C}}$, while in $\mathbf{Hybrid}_5$, these functionalities use the values computed by $\mathcal{S}$. Therefore, the distribution of $\mathbf{Hybrid}_5$ is the same as the distribution of $\mathbf{Hybrid}_4$.

$\mathbf{Hybrid}_6$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\text{input-mal}}$ as above. The only difference is that $\mathcal{S}$ does not generate the shares of honest parties, which are not used in the future steps in $\mathbf{Hybrid}_5$. Therefore, the distribution of $\mathbf{Hybrid}_6$ is the same as the distribution of $\mathbf{Hybrid}_5$.

Note that $\textbf{Hybrid}_6$ is the execution in the ideal world, and the distribution of $\textbf{Hybrid}_6$ is identical to the distribution of $\textbf{Hybrid}_0$, the execution in the real world. $\square$

Analysis of the Communication Complexity of Main. Compared with the semi-honest protocol MAIN-SEMI, all parties only invoke two additional functionalities $\mathcal{F}_{\text{mult-veri}}, \mathcal{F}_{\text{network}}$. For $\mathcal{F}_{\text{mult-veri}}$, recall that the communication complexity of its implementation MULT-VERI is $O(n^3 \cdot \kappa^2)$ bits. For $\mathcal{F}_{\text{network}}$, recall that the communication complexity of its implementation NETWORK is $O(n^5 \cdot \kappa^2)$ bits. Thus, the communication complexity of MAIN is

$$O(|C| \cdot n/k + n \cdot (c + \mathsf{Depth}) + n^5 \cdot \kappa^2)$$

field elements.

Theorem 8. *In the client-server model, let c denote the number of clients, and $n = 2t+1$ denote the number of parties (servers). Let κ denote the security parameter, and $\mathbb{F}$ denote a finite field. For an arithmetic circuit C over $\mathbb{F}$ and for all $1 \leq k \leq t$, 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 $t-k+1$ parties. The communication complexity of this protocol is $O(|C| \cdot n/k + n \cdot (c + \mathsf{Depth}) + n^5 \cdot \kappa^2)$ elements in $\mathbb{F}$.*

References

[BBCG$^+$19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 67–97, Cham, 2019. Springer International Publishing.

[BGIN20] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Efficient fully secure computation via distributed zero-knowledge proofs. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 244–276, Cham, 2020. Springer International Publishing.

[BGJK21] Gabrielle Beck, Aarushi Goel, Abhishek Jain, and Gabriel Kaptchuk. Order-c secure multiparty computation for highly repetitive circuits. Appears in Eurocrypt, 2021. https://eprint.iacr.org/2021/500.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10. ACM, 1988.

[BSFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 663–680, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In Ran Canetti, editor, *Theory of Cryptography*, pages 213–230, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[CCD88] 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.

[CCXY18] Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized complexity of information-theoretically secure mpc revisited. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 395–426, Cham, 2018. Springer International Publishing.

[CGH$^+$18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority mpc for malicious adversaries. In *Annual International Cryptology Conference*, pages 34–64. Springer, 2018.

[DIK10] 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.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*, pages 572–590. Springer, 2007.

[FY92] 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 699–710, New York, NY, USA, 1992. Association for Computing Machinery.

[GIOZ17] Juan Garay, Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. The price of low communication in secure multi-party computation. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 420–446, Cham, 2017. Springer International Publishing.

[GIP$^+$14] 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.

[GIP15] 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.

[GLS19] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional mpc with guaranteed output delivery. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 85–114, Cham, 2019. Springer International Publishing.

[GMW87] 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.

[GS20] 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.

[GSY21] S. Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The more the merrier: Reducing the cost of large scale mpc. Appears in Eurocrypt, 2021. https://eprint.iacr.org/2021/303.

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority mpc. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 618–646, Cham, 2020. Springer International Publishing.

[HM01] Martin Hirt and Ueli Maurer. Robustness for free in unconditional multi-party computation. In *Annual International Cryptology Conference*, pages 101–118. Springer, 2001.

[HMP00] 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.

[NV18] Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority mpc by batchwise multiplication verification. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security*, pages 321–339, Cham, 2018. Springer International Publishing.

[PS21] Antigoni Polychroniadou and Yifan Song. Constant-overhead unconditionally secure multiparty computation over binary fields. Appears in Eurocrypt, 2021. https://eprint.iacr.org/2020/1412.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[Yao82] Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

A Realizing $\mathcal{F}_{\text{rand}}$

In this section, we introduce the protocol for $\mathcal{F}_{\text{rand}}$. For a general $\mathbb{F}$-linear secret sharing scheme Σ, let $[\![x]\!]$ denote a sharing in Σ of secret x. For a set $A \subset \mathcal{I}$, recall that $\pi_A([\![x]\!])$ refers to the shares of $[\![x]\!]$ held by parties in A. We assume that Σ satisfies the following property:

- Given a set $A \subset \mathcal{I}$ and a set of shares $\{a_i\}_{i\in A}$ for parties in A, let

$$\Sigma(A, (a_i)_{i\in A}) := \{[\![x]\!] |\ [\![x]\!] \in \Sigma \text{ and } \pi_A([\![x]\!]) = (a_i)_{i\in A}\}.$$

  Then, there is an efficient algorithm which outputs that either $\Sigma(A, (a_i)_{i\in A}) = \emptyset$, or a random sharing $[\![x]\!]$ in $\Sigma(A, (a_i)_{i\in A})$.

We first introduce a tensoring-up lemma from [CCXY18].

A Tensoring-up Lemma. We follow the definition of interleaved GLSSS: the m-fold interleaved GLSSS $\Sigma^{\times m}$ is an n-party scheme which corresponds to m Σ-sharings. We have the following proposition from [CCXY18]:

Proposition 1 ([CCXY18])**.** *Let $\mathbb{G}$ be a degree-m extension field of $\mathbb{F}$ and let Σ be a $\mathbb{F}$-GLSSS. Then the m-fold interleaved $\mathbb{F}$-GLSSS $\Sigma^{\times m}$ is naturally viewed as an $\mathbb{G}$-GLSSS, compatible with its $\mathbb{F}$-linearity.*

This proposition allows us to define $\lambda : \Sigma^{\times m} \to \Sigma^{\times m}$ for every $\lambda \in \mathbb{G}$ 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 $\lambda_1, \lambda_2 \in \mathbb{G}$, $\lambda_1 \cdot [\![\boldsymbol{x}]\!] + \lambda_2 \cdot [\![\boldsymbol{x}]\!] = (\lambda_1 + \lambda_2) \cdot [\![\boldsymbol{x}]\!]$;
- for all $\lambda_1, \lambda_2 \in \mathbb{G}$, $\lambda_1 \cdot (\lambda_2 \cdot [\![\boldsymbol{x}]\!]) = (\lambda_1 \cdot \lambda_2) \cdot [\![\boldsymbol{x}]\!]$.

An Example of a GLSSS and Using the Tensoring-up Lemma. We will use the standard Shamir secret sharing scheme as an example of a GLSSS 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. The secret space and the share space of Σ are $\mathbb{F}$. According to the Lagrange interpolation, the secret x can be written as a $\mathbb{F}$-linear combination of all the shares. Therefore, the defining map of Σ is $\mathbb{F}$-linear. Thus Σ is a $\mathbb{F}$-GLSSS.

A sharing $[\boldsymbol{x}]_t = ([x_1]_t, [x_2]_t, \ldots, [x_m]_t) \in \Sigma^{\times m}$ is a vector of m sharings in Σ. Let $\mathbb{G}$ be a degree-m extension field of $\mathbb{F}$. The tensoring-up lemma says that $\Sigma^{\times m}$ is a $\mathbb{G}$-GLSSS. Therefore we can perform $\mathbb{G}$-linear operations to the sharings in $\Sigma^{\times m}$.

A High-level Idea of the Construction. As [PS21], we will follow the idea in [BSFO12] 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 $\mathbb{G}$ be an extension field of $\mathbb{F}$ such that $|\mathbb{G}| \geq 2^\kappa$. Let $m = [\mathbb{G} : \mathbb{F}]$ 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 1, $\Sigma^{\times m}$ is an $\mathbb{G}$-GLSSS.

A.1 Preparing Verified Sharings.

The first step is to let each party deal a batch of random sharings in $\Sigma^{\times m}$. The protocol $\textsc{VerShare}(P_d, N')$ (Protocol 49) allows the dealer P_d to deal N' random sharings in $\Sigma^{\times m}$. Suppose the share size of a sharing in Σ is sh field elements in $\mathbb{F}$. Then the share size of a sharing in $\Sigma^{\times m}$ is $m \cdot \mathsf{sh}$ field elements in $\mathbb{F}$. Recall that the communication complexity of the instantiation of $\mathcal{F}_{\text{coin}}(\mathbb{G})$ in [GSZ20] is $O(n^2)$ elements in $\mathbb{G}$, which is $O(n^2 \cdot m)$ elements in $\mathbb{G}$. The communication complexity of $\textsc{VerShare}(P_d, N')$ is $O(N' \cdot n \cdot m \cdot \mathsf{sh} + n^2 \cdot m)$ elements in $\mathbb{F}$.

Protocol 49: $\textsc{VerShare}(P_d, N')$

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 $\mathcal{F}_{\text{coin}}(\mathbb{G})$ and receive a random field element $\alpha \in \mathbb{G}$. 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.

For a sharing $[\![s]\!]$ and a nonempty set A, 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 27. *Let $\mathcal{H} \subset \mathcal{I}$ denote the set of all honest parties. If all parties accept the verification in the last step of* $\textsc{VerShare}$*(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 $N'/2^\kappa$.*

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 $\alpha \in \mathbb{G}$ such that $[\![\boldsymbol{s}]\!]$ passes the verification in the last step of $\textsc{VerShare}(P_d, N')$ is bounded by N'. Then, the lemma follows from that α output by $\mathcal{F}_{\text{coin}}$ is uniformly random in $\mathbb{G}$ and $|\mathbb{G}| \geq 2^\kappa$. 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 $\mathbb{G}$ 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 $\mathbb{G}$-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. □

A.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 [BSFO12], 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 50.

Protocol 50: 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 $\mathbb{G}$. 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 Rand(N) (Protocol 51) which securely computes $\mathcal{F}_{\mathrm{rand}}$. Compared with [PS21], we show that this construction is secure for any corruption threshold $t' < n$. However, the overall communication complexity will depend on the corruption threshold. We will analyze the communication complexity after the proof of Lemma 28.

Protocol 51: 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.

Lemma 28. *Let $t' < n$ be a positive integer. Protocol* Rand *securely computes $\mathcal{F}_{rand}$ (Functionality 2) with abort in the $\mathcal{F}_{coin}$-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 $\mathcal{C}$ 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 $\pi_{\mathcal{C}}([\![\boldsymbol{s}^{(\ell)}]\!])$ to $\mathcal{A}$.
- In Step 3, $\mathcal{S}$ emulates $\mathcal{F}_{\text{coin}}$ and generates a random field element $\alpha \in \mathbb{G}$. Then, $\mathcal{S}$ computes the shares of $[\![\boldsymbol{s}]\!]$ held by corrupted parties, i.e., $\pi_{\mathcal{C}}([\![\boldsymbol{s}]\!])$. Based on $\pi_{\mathcal{C}}([\![\boldsymbol{s}]\!])$, $\mathcal{S}$ randomly samples $[\![\boldsymbol{s}]\!] \in \Sigma^{\times m}(\mathcal{C}, \pi_{\mathcal{C}}([\![\boldsymbol{s}]\!]))$.
- 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 $\mathcal{F}_{\text{rand}}$ 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 $\mathcal{F}_{\text{rand}}$ and aborts.

Simulation for Convert. Recall that Convert only involves local computation. For each sharing $[\![\boldsymbol{s}_d]\!]$, $\mathcal{S}$ has computed $\pi_{\mathcal{C}}([\![\boldsymbol{s}_d]\!])$ in the simulation of VerShare. In Convert, $\mathcal{S}$ computes $\pi_{\mathcal{C}}([\![\boldsymbol{r}_i]\!])$ and sends them to $\mathcal{F}_{\text{rand}}$. Then $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\text{rand}}$.

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 27, 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 $[\![\tilde{\boldsymbol{s}}^{(\ell)}]\!] \in \Sigma^{\times m}(\mathcal{C}, \pi_{\mathcal{C}}([\![\boldsymbol{s}^{(\ell)}]\!]))$. $\mathcal{S}$ 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 $\pi_{\mathcal{C}}([\![\boldsymbol{s}^{(\ell)}]\!])$. Therefore, $[\![\boldsymbol{s}^{(\ell)}]\!]$ is a random sharing in $\Sigma^{\times m}(\mathcal{C}, \pi_{\mathcal{C}}([\![\boldsymbol{s}^{(\ell)}]\!]))$. In Step 3, after receiving $\alpha \in \mathbb{G}$ 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 $\Sigma^{\times m}(\mathcal{C}, \pi_{\mathcal{C}}([\![\boldsymbol{s}]\!]))$. 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 $\Sigma^{\times m}(\mathcal{C}, \pi_{\mathcal{C}}([\![\boldsymbol{s}^{(\ell)}]\!]))$. This means that $\mathcal{S}$ can re-sample and use a new random sharing $[\![\tilde{\boldsymbol{s}}^{(\ell)}]\!] \in \Sigma^{\times m}(\mathcal{C}, \pi_{\mathcal{C}}([\![\boldsymbol{s}^{(\ell)}]\!]))$ instead of using the sharing $[\![\boldsymbol{s}^{(\ell)}]\!]$ generated in the beginning.

Thus, the distribution of **Hybrid**$_2$ is the same as the distribution of **Hybrid**$_1$.

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 $\boldsymbol{M}_{\mathcal{C}}$ denote the sub-matrix of $\boldsymbol{M}$ containing columns with indices in $\mathcal{C}$, and $\boldsymbol{M}_{\mathcal{H}}$ denote the sub-matrix containing columns with indices in $\mathcal{H}$. We have

$$([\![\boldsymbol{r}_1]\!], \dots, [\![\boldsymbol{r}_{n-t'}]\!])^{\mathrm{T}} = \boldsymbol{M} \cdot ([\![\boldsymbol{s}_1]\!], \dots, [\![\boldsymbol{s}_n]\!])^{\mathrm{T}} = \boldsymbol{M}_{\mathcal{C}} \cdot ([\![\boldsymbol{s}_j]\!])^{\mathrm{T}}_{j\in\mathcal{C}} + \boldsymbol{M}_{\mathcal{H}} \cdot ([\![\boldsymbol{s}_j]\!])^{\mathrm{T}}_{j\in\mathcal{H}}.$$

Let

$$\begin{aligned} ([\![\boldsymbol{r}_1^{(\mathcal{H})}]\!], \dots, [\![\boldsymbol{r}_{n-t'}^{(\mathcal{H})}]\!])^{\mathrm{T}} &:= \boldsymbol{M}_{\mathcal{H}} \cdot ([\![\boldsymbol{s}_j]\!])^{\mathrm{T}}_{j\in\mathcal{H}} \\ ([\![\boldsymbol{r}_1^{(\mathcal{C})}]\!], \dots, [\![\boldsymbol{r}_{n-t'}^{(\mathcal{C})}]\!])^{\mathrm{T}} &:= \boldsymbol{M}_{\mathcal{C}} \cdot ([\![\boldsymbol{s}_j]\!])^{\mathrm{T}}_{j\in\mathcal{C}}. \end{aligned}$$

Then $([\![\boldsymbol{r}_1]\!], \dots, [\![\boldsymbol{r}_{n-t'}]\!]) = ([\![\boldsymbol{r}_1^{(\mathcal{C})}]\!], \dots, [\![\boldsymbol{r}_{n-t'}^{(\mathcal{C})}]\!]) + ([\![\boldsymbol{r}_1^{(\mathcal{H})}]\!], \dots, [\![\boldsymbol{r}_{n-t'}^{(\mathcal{H})}]\!])$.

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 $([\![\boldsymbol{r}_1^{(\mathcal{H})}]\!], \dots, [\![\boldsymbol{r}_{n-t'}^{(\mathcal{H})}]\!])$ to $([\![\boldsymbol{s}_j]\!])_{j\in\mathcal{H}}$. Given $([\![\boldsymbol{s}_j]\!])_{j\in\mathcal{C}}$, there is a one-to-one map from $([\![\boldsymbol{r}_1]\!], \dots, [\![\boldsymbol{r}_{n-t'}]\!])$ to $([\![\boldsymbol{r}_1^{(\mathcal{H})}]\!], \dots, [\![\boldsymbol{r}_{n-t'}^{(\mathcal{H})}]\!])$. 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 $\mathcal{F}_{\mathrm{rand}}$.

To see this, note that for each valid $([\![\tilde{\boldsymbol{s}}_j]\!])_{j\in\mathcal{C}}$, $\mathcal{S}$ computes $\{\pi_{\mathcal{C}}([\![\boldsymbol{r}_j]\!])\}_{j=1}^{n-t'}$ and sends them to $\mathcal{F}_{\mathrm{rand}}$. Then for each $j \in \{1, \dots, n-t'\}$, $\mathcal{F}_{\mathrm{rand}}$ samples a random sharing $[\![\tilde{\boldsymbol{r}}_j]\!] \in \Sigma^{\times m}(\mathcal{C}, \pi_{\mathcal{C}}([\![\boldsymbol{r}_j]\!]))$. 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 $([\![\tilde{\boldsymbol{s}}_j]\!])_{j\in\mathcal{C}}$. 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. $\square$

Analysis of the Communication Complexity of 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 $O(N' \cdot n^2 \cdot m \cdot \mathsf{sh} + n^3 \cdot m)$ elements in $\mathbb{F}$. Step 2 and Step 3 do not require any communication. Recall that $N' = \frac{N}{m(n-t')}$ and $m = [\mathbb{G} : \mathbb{F}]$ is bounded by the security parameter κ. Therefore, the overall communication complexity of Rand(N) is

$$O(N' \cdot n^2 \cdot m \cdot \mathsf{sh} + n^3 \cdot m) = O(N \cdot n^2/(n-t') \cdot \mathsf{sh} + n^3 \cdot \kappa)$$

elements in $\mathbb{F}$.

In this work, we always have $t' \le \frac{n-1}{2}$. Therefore, $n/(n-t')$ is a constant. The communication complexity of generating N random sharings in Σ is $O(N \cdot n \cdot \mathsf{sh} + n^3 \cdot \kappa)$ elements in $\mathbb{F}$.

B Using $\mathcal{F}_{\mathrm{rand}}$ to Prepare Various Random Sharings

B.1 Preparing Random Sharings in the form of $([\boldsymbol{r}], \langle \boldsymbol{r} \rangle)$

Consider a secret sharing scheme Σ which takes a vector $\boldsymbol{x} \in \mathbb{F}^k$ and outputs a pair of packed Shamir sharing $([\boldsymbol{x}], \langle \boldsymbol{x} \rangle)$. Note that the packed Shamir sharing scheme is linear in $\mathbb{F}$. Therefore Σ is an $\mathbb{F}$-GLSSS. To use $\mathcal{F}_{\mathrm{rand}}$ to prepare random sharings in Σ, we need to show that:

- Given a set $A \subset \mathcal{I}$ and a set of shares $\{a_i\}_{i\in A}$ for parties in A, there exists an efficient algorithm which outputs that either $\Sigma(A, (a_i)_{i\in A}) = \emptyset$, or a random sharing $([\boldsymbol{x}], \langle \boldsymbol{x} \rangle) \in \Sigma(A, (a_i)_{i\in A})$. Note that each share a_i is a pair of elements $(a_i^{(0)}, a_i^{(1)})$ in $\mathbb{F}$.

Depending on the size of A, there are two cases:

- If $|A| \geq t+1$, by the property of the packed Shamir sharing scheme, the set of shares $\{a_i^{(0)}\}_{i \in A}$ can fully determine the whole sharing $[\boldsymbol{x}]$ if exists. The algorithm checks whether the shares $\{a_i^{(0)}\}_{i \in A}$ lie on a polynomial $f(\cdot)$ of degree at most t in $\mathbb{F}$. If not, the algorithm outputs $\Sigma(A, (a_i)_{i \in A}) = \emptyset$. Otherwise, the algorithm recovers the whole sharing $[\boldsymbol{x}]$ and reconstructs the secrets $\boldsymbol{x}$.
  - If $|A| \geq 2t+1-k$, by the property of the packed Shamir sharing scheme, the set of shares $\{a_i^{(1)}\}_{i \in A}$ and the secrets $\boldsymbol{x}$ can fully determine the whole sharing $\langle \boldsymbol{x} \rangle$ if exists. The algorithm checks whether the shares $\{a_i^{(1)}\}_{i \in A}$ and the secrets $\boldsymbol{x}$ lie on a polynomial $g(\cdot)$ of degree at most $2t$ in $\mathbb{F}$. If not, the algorithm outputs $\Sigma(A, (a_i)_{i \in A}) = \emptyset$. Otherwise, the algorithm recovers the whole sharing $\langle \boldsymbol{x} \rangle$ and outputs $([\boldsymbol{x}], \langle \boldsymbol{x} \rangle)$.
  - If $|A| < 2t+1-k$, by the property of the packed Shamir sharing scheme, the set of shares $\{a_i^{(1)}\}_{i \in A}$ is independent of the secret. The algorithm randomly samples $2t+1-k-|A|$ elements in $\mathbb{F}$ as the shares of the first $2t+1-k-|A|$ parties in $\mathcal{I} \backslash A$. Then, based on the secrets $\boldsymbol{x}$, the shares of the first $2t+1-k-|A|$ parties in $\mathcal{I} \backslash A$, and the shares $\{a_i^{(1)}\}_{i \in A}$, the algorithm reconstructs the whole packed Shamir sharing $\langle \boldsymbol{x} \rangle$ and outputs $([\boldsymbol{x}], \langle \boldsymbol{x} \rangle)$.
- If $|A| \leq t$, by the property of the packed Shamir sharing scheme, $[\boldsymbol{x}]$ needs additional $t+1-|A|$ shares to be determined. The algorithm randomly samples $t+1-|A|$ elements in $\mathbb{F}$ as the shares of the first $t+1-|A|$ parties in $\mathcal{I} \backslash A$. Then, based on the shares of the first $t+1-|A|$ parties in $\mathcal{I} \backslash A$ and the shares $\{a_i^{(0)}\}_{i \in A}$, the algorithm reconstructs the whole packed Shamir sharing $[\boldsymbol{x}]$ and the secrets $\boldsymbol{x}$. For $\langle \boldsymbol{x} \rangle$, it needs additional $2t+1-k-|A|$ shares to be determined since we already have $|A|$ shares $\{a_i^{(1)}\}_{i \in A}$ and the secrets $\boldsymbol{x}$. The algorithm randomly samples $2t+1-k-|A|$ elements in $\mathbb{F}$ as the shares of the first $2t+1-k-|A|$ parties in $\mathcal{I} \backslash A$. Then, based on the secrets $\boldsymbol{x}$, the shares of the first $2t+1-k-|A|$ parties in $\mathcal{I} \backslash A$, and the shares $\{a_i^{(1)}\}_{i \in A}$, the algorithm reconstructs the whole packed Shamir sharing $\langle \boldsymbol{x} \rangle$ and outputs $([\boldsymbol{x}], \langle \boldsymbol{x} \rangle)$.

Thus, we can use $\mathcal{F}_{\text{rand}}$ to prepare random sharings in Σ defined above. Note that the share size of a sharing in Σ is $\mathsf{sh} = 2$ elements in $\mathbb{F}$. Therefore, the communication complexity of generating m random sharings in Σ is $O(m \cdot n + n^3 \cdot \kappa)$ elements in $\mathbb{F}$.

B.2 Preparing Random Sharings in the form of $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}])$ for a Fixed Permutation $p(\cdot)$

Consider a secret sharing scheme Σ which takes a vector $\boldsymbol{r} \in \mathbb{F}^k$ and outputs a pair of two degree-t packed Shamir sharings $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}])$. Note that both parts are linear in $\mathbb{F}$. Therefore Σ is an $\mathbb{F}$-GLSSS. To use $\mathcal{F}_{\text{rand}}$ to prepare random sharings in Σ, we need to show that:

- Given a set $A \subset \mathcal{I}$ and a set of shares $\{a_i\}_{i \in A}$ for parties in A, there exists an efficient algorithm which outputs that either $\Sigma(A, (a_i)_{i \in A}) = \emptyset$, or a random sharing $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}]) \in \Sigma(A, (a_i)_{i \in A})$. Note that each share a_i is a pair of elements $(a_i^{(0)}, a_i^{(1)})$ in $\mathbb{F}$.

Depending on the size of A, there are three cases:

- If $|A| \geq t+1$, by the property of the degree-t packed Shamir sharing scheme, the set of shares $\{a_i\}_{i \in A}$ can fully determine the whole sharings if exist. The algorithm parses each share a_i to $(a_i^{(0)}, a_i^{(1)})$. Then, the algorithm checks whether the shares $\{a_i^{(0)}\}_{i \in A}$ lie on a polynomial $f(\cdot)$ of degree at most t in $\mathbb{F}$, and, the shares $\{a_i^{(1)}\}_{i \in A}$ lie on a polynomial $g(\cdot)$ of degree at most t in $\mathbb{F}$. If not, the algorithm outputs $\Sigma(A, (a_i)_{i \in A}) = \emptyset$. Otherwise, the algorithm sets $[\boldsymbol{r}]$ to be the sharing determined by $f(\cdot)$ and $[\boldsymbol{r}']$ to be the sharing determined by $g(\cdot)$. The algorithm checks whether $\boldsymbol{r}' = \boldsymbol{M}_p \cdot \boldsymbol{r}$. If not, the algorithm outputs $\Sigma(A, (a_i)_{i \in A}) = \emptyset$. Otherwise, the algorithm outputs $([\boldsymbol{r}], [\boldsymbol{r}'])$.

- If $t-k+1 < |A| \leq t$, the set of shares $\{a_i\}_{i\in A}$ do not determine the whole sharings but are correlated with the secrets. The algorithm parses each share a_i to $(a_i^{(0)}, a_i^{(1)})$. To determine the whole sharings, we need additional $t+1-|A|$ shares. Let B denote the set of the first $t+1-|A|$ parties in $\mathcal{I}\backslash A$, and let $\{(x_i^{(0)}, x_i^{(1)})\}_{i\in B}$ be the variables of the shares of parties in B. Then the secrets $\boldsymbol{r}$ of the sharing determined by the shares $\{a_i^{(0)}\}_{i\in A}\bigcup\{x_i^{(0)}\}_{i\in B}$ can be expressed by linear combinations of $\{a_i^{(0)}\}_{i\in A}\bigcup\{x_i^{(0)}\}_{i\in B}$. Similarly, the secrets $\boldsymbol{r}'$ of the sharing determined by the shares $\{a_i^{(1)}\}_{i\in A}\bigcup\{x_i^{(1)}\}_{i\in B}$ can be expressed by linear combinations of $\{a_i^{(1)}\}_{i\in A}\bigcup\{x_i^{(1)}\}_{i\in B}$. The requirement $\boldsymbol{r}' = \boldsymbol{M}_p \cdot \boldsymbol{r}$ establishes a set of linear equations among the variables $\{x_i^{(0)}, x_i^{(1)}\}_{i\in B}$. It is well known that linear equations can be solved within polynomial time, and it is possible to compute a random solution if exists. The algorithm checks whether there exists a solution. If not, the algorithm outputs $\Sigma(A, (a_i)_{i\in A}) = \emptyset$. Otherwise, the algorithm computes a random solution. Let $[\boldsymbol{r}]$ denote the sharing determined by the shares $\{a_i^{(0)}\}_{i\in A}\bigcup\{x_i^{(0)}\}_{i\in B}$, and $[\boldsymbol{r}']$ denote the sharing determined by the shares $\{a_i^{(1)}\}_{i\in A}\bigcup\{x_i^{(1)}\}_{i\in B}$. The algorithm outputs $([\boldsymbol{r}], [\boldsymbol{r}'])$.
- If $|A| \leq t-k+1$, by the property of the degree-t packed Shamir sharing scheme, the set of shares $\{a_i\}_{i\in A}$ are independent of the secrets. Therefore, $\Sigma(A, (a_i)_{i\in A}) \neq \emptyset$. The algorithm randomly samples $\boldsymbol{r} \in \mathbb{F}^k$ and randomly samples $t-k+1-|A|$ pairs of elements in $\mathbb{F}$ as the shares of the first $t-k+1-|A|$ parties in $\mathcal{I}\backslash A$. Then, based on the secrets $\boldsymbol{r}, \boldsymbol{M}_p \cdot \boldsymbol{r}$, the shares of the first $t-k+1-|A|$ parties in $\mathcal{I}\backslash A$, and the shares $\{a_i\}_{i\in A}$, the algorithm reconstructs the two degree-t packed Shamir sharings $[\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}]$ and outputs $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}])$.

Thus, we can use $\mathcal{F}_{\text{rand}}$ with the secret sharing scheme Σ to prepare random sharings $([\boldsymbol{r}], [\boldsymbol{M}_p \cdot \boldsymbol{r}])$. Note that the share size of a sharing in Σ is $\mathsf{sh} = 2$ elements in $\mathbb{F}$. Therefore, the communication complexity of preparing m random sharings in Σ is $O(m\cdot n + n^3 \cdot \kappa)$ elements in $\mathbb{F}$.

B.3 Preparing Random degree-$2t$ packed Shamir sharings of 0

Consider a secret sharing scheme Σ which outputs a degree-$2t$ packed Shamir sharing of $\mathbf{0} \in \mathbb{F}^k$. The secret space of Σ is $\{\mathbf{0}\}$. It is clear that Σ is an $\mathbb{F}$-GLSSS. To use $\mathcal{F}_{\text{rand}}$ to prepare random sharings in Σ, we need to show that:

- Given a set $A \subset \mathcal{I}$ and a set of shares $\{a_i\}_{i\in A}$ for parties in A, there exists an efficient algorithm which outputs that either $\Sigma(A, (a_i)_{i\in A}) = \emptyset$, or a random sharing $\langle \mathbf{0} \rangle \in \Sigma(A, (a_i)_{i\in A})$.

Depending on the size of A, there are two cases:

- If $|A| \geq 2t+1-k$, then the whole sharing is determined by $\{a_i\}_{i\in A}$ and the secrets $\mathbf{0}$ if exists. The algorithm checks whether there exists a degree-$2t$ polynomial $f(\cdot)$ in $\mathbb{F}$ such that for all $i \in A$, $f(\alpha_i) = a_i$, and for all $i \in [k]$, $f(\beta_i) = 0$. If not, the algorithm outputs $\Sigma(A, (a_i)_{i\in A}) = \emptyset$. Otherwise, the algorithm sets $\langle \mathbf{0} \rangle$ to be the sharing determined by $f(\cdot)$ and outputs $\langle \mathbf{0} \rangle$.
- If $|A| < 2t+1-k$, then the set $\Sigma(A, (a_i)_{i\in A})$ is non-empty. The algorithm randomly samples $2t+1-k-|A|$ elements in $\mathbb{F}$ as the shares for the first $2t+1-k-|A|$ parties in $\mathcal{I}\backslash A$. Based on the secrets $\mathbf{0}$, the shares of the first $2t+1-k-|A|$ parties in $\mathcal{I}\backslash A$, and the shares $\Sigma(A, (a_i)_{i\in A})$, the algorithm reconstructs the whole sharing $\langle \mathbf{0} \rangle$ and outputs $\langle \mathbf{0} \rangle$.

Thus, we can use $\mathcal{F}_{\text{rand}}$ with the secret sharing scheme Σ to prepare random degree-$2t$ packed Shamir sharings of $\mathbf{0}$. Note that the share size of a sharing in Σ is $\mathsf{sh} = 1$ element in $\mathbb{F}$. Therefore, the communication complexity of preparing m random sharings in Σ is $O(m\cdot n + n^3 \cdot \kappa)$ elements in $\mathbb{F}$.

B.4 Preparing Random Sharings for a Fixed Pattern π

Recall that, for all $i_1, i_2 \in \{1, 2, \ldots, k\}$, we say a pair of degree-t packed Shamir sharings $([\boldsymbol{x}], [\boldsymbol{y}])$ contains an (i_1, i_2)-block if for all $i_1 \leq j \leq i_2$ the secrets of these two sharings satisfy that $y_j = x_{i_1}$. We say a point

$j \in \{1, 2, \dots, k\}$ is covered by an (i_1, i_2)-block if $i_1 \leq j \leq i_2$. A pattern π is defined to be a list of blocks such that for all $j \in \{1, 2, \dots, k\}$, j is covered by exactly one block in π. For a fixed pattern π, our goal is to prepare a pair of random sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ such that for all (i_1, i_2)-block in π and $i_1 \leq j \leq i_2$, $\tilde{r}_j = r_{i_1}$.

Consider a secret sharing scheme Σ which takes a vector $\boldsymbol{r} \in \mathbb{F}^k$ and outputs a pair of two degree-t packed Shamir sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ such that for all (i_1, i_2)-block in π and $i_1 \leq j \leq i_2$, $\tilde{r}_j = r_{i_1}$. Note that both parts are linear in $\mathbb{F}$. Therefore Σ is an $\mathbb{F}$-GLSSS. To use $\mathcal{F}_{\text{rand}}$ to prepare random sharings in Σ, we need to show that:

- Given a set $A \subset \mathcal{I}$ and a set of shares $\{a_i\}_{i \in A}$ for parties in A, there exists an efficient algorithm which outputs that either $\Sigma(A, (a_i)_{i \in A}) = \emptyset$, or a random sharing $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}]) \in \Sigma(A, (a_i)_{i \in A})$. Note that each share a_i is a pair of elements $(a_i^{(0)}, a_i^{(1)})$ in $\mathbb{F}$.

Depending on the size of A, there are three cases:

- If $|A| \geq t + 1$, by the property of the degree-t packed Shamir sharing scheme, the set of shares $\{a_i\}_{i \in A}$ can fully determine the whole sharings if exist. The algorithm parses each share a_i to $(a_i^{(0)}, a_i^{(1)})$. Then, the algorithm checks whether the shares $\{a_i^{(0)}\}_{i \in A}$ lie on a polynomial $f(\cdot)$ of degree at most t in $\mathbb{F}$, and, the shares $\{a_i^{(1)}\}_{i \in A}$ lie on a polynomial $g(\cdot)$ of degree at most t in $\mathbb{F}$. If not, the algorithm outputs $\Sigma(A, (a_i)_{i \in A}) = \emptyset$. Otherwise, the algorithm sets $[\boldsymbol{r}]$ to be the sharing determined by $f(\cdot)$ and $[\tilde{\boldsymbol{r}}]$ to be the sharing determined by $g(\cdot)$. The algorithm checks whether for all (i_1, i_2)-block in π and $i_1 \leq j \leq i_2$, $\tilde{r}_j = r_{i_1}$. If not, the algorithm outputs $\Sigma(A, (a_i)_{i \in A}) = \emptyset$. Otherwise, the algorithm outputs $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$.
- If $t - k + 1 < |A| \leq t$, the set of shares $\{a_i\}_{i \in A}$ do not determine the whole sharings but are correlated with the secrets. The algorithm parses each share a_i to $(a_i^{(0)}, a_i^{(1)})$. To determine the whole sharings, we need additional $t + 1 - |A|$ shares. Let B denote the set of the first $t + 1 - |A|$ parties in $\mathcal{I} \backslash A$, and let $\{(x_i^{(0)}, x_i^{(1)})\}_{i \in B}$ be the variables of the shares of parties in B. Then the secrets $\boldsymbol{r}$ of the sharing determined by the shares $\{a_i^{(0)}\}_{i \in A} \bigcup \{x_i^{(0)}\}_{i \in B}$ can be expressed by linear combinations of $\{a_i^{(0)}\}_{i \in A} \bigcup \{x_i^{(0)}\}_{i \in B}$. Similarly, the secrets $\tilde{\boldsymbol{r}}$ of the sharing determined by the shares $\{a_i^{(1)}\}_{i \in A} \bigcup \{x_i^{(1)}\}_{i \in B}$ can be expressed by linear combinations of $\{a_i^{(1)}\}_{i \in A} \bigcup \{x_i^{(1)}\}_{i \in B}$. The requirement that for all (i_1, i_2)-block in π and $i_1 \leq j \leq i_2$, $\tilde{r}_j = r_{i_1}$ establishes a set of linear equations among the variables $\{x_i^{(0)}, x_i^{(1)}\}_{i \in B}$. It is well known that linear equations can be solved within polynomial time, and it is possible to compute a random solution if exists. The algorithm checks whether there exists a solution. If not, the algorithm outputs $\Sigma(A, (a_i)_{i \in A}) = \emptyset$. Otherwise, the algorithm computes a random solution. Let $[\boldsymbol{r}]$ denote the sharing determined by the shares $\{a_i^{(0)}\}_{i \in A} \bigcup \{x_i^{(0)}\}_{i \in B}$, and $[\tilde{\boldsymbol{r}}]$ denote the sharing determined by the shares $\{a_i^{(1)}\}_{i \in A} \bigcup \{x_i^{(1)}\}_{i \in B}$. The algorithm outputs $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$.
- If $|A| \leq t - k + 1$, by the property of the degree-t packed Shamir sharing scheme, the set of shares $\{a_i\}_{i \in A}$ are independent of the secrets. Therefore, $\Sigma(A, (a_i)_{i \in A}) \neq \emptyset$. The algorithm randomly samples $\boldsymbol{r} \in \mathbb{F}^k$ and and computes a vector $\tilde{\boldsymbol{r}}$ such that for all (i_1, i_2)-block in π and $i_1 \leq j \leq i_2$, $\tilde{r}_j = r_{i_1}$. The algorithm also randomly samples $t - k + 1 - |A|$ pairs of elements in $\mathbb{F}$ as the shares of the first $t - k + 1 - |A|$ parties in $\mathcal{I} \backslash A$. Then, based on the secrets $\boldsymbol{r}, \tilde{\boldsymbol{r}}$, the shares of the first $t - k + 1 - |A|$ parties in $\mathcal{I} \backslash A$, and the shares $\{a_i\}_{i \in A}$, the algorithm reconstructs the two degree-t packed Shamir sharings $[\boldsymbol{r}], [\tilde{\boldsymbol{r}}]$ and outputs $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$.

Thus, we can use $\mathcal{F}_{\text{rand}}$ with the secret sharing scheme Σ to prepare random sharings $([\boldsymbol{r}], [\tilde{\boldsymbol{r}}])$ for a fixed pattern π. Note that the share size of a sharing in Σ is $\mathsf{sh} = 2$ elements in $\mathbb{F}$. Therefore, the communication complexity of preparing m random sharings in Σ is $O(m \cdot n + n^3 \cdot \kappa)$ elements in $\mathbb{F}$.

B.5 Preparing Random Wire Tuples for Fixed $i, j \in \{1, 2, \dots, k\}$

Recall that a wire tuple $([\boldsymbol{x}], [\boldsymbol{y}], i, j)$ satisfies that $x_i = y_j$. Our goal is to prepare a random wire tuple for fixed i, j.

Consider a secret sharing scheme Σ which takes a pair of elements $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{K}^2$ such that $x_i = y_j$ and outputs a pair of two degree-t packed Shamir sharings $([\boldsymbol{x}], [\boldsymbol{y}]$. Note that the secret space is $\mathbb{K}$-linear and both parts of the output are linear in $\mathbb{K}$. Therefore Σ is an $\mathbb{K}$-GLSSS. To use $\mathcal{F}_{\text{rand}}$ to prepare random sharings in Σ, we need to show that:

- Given a set $A \subset \mathcal{I}$ and a set of shares $\{a_i\}_{i\in A}$ for parties in A, there exists an efficient algorithm which outputs that either $\Sigma(A, (a_i)_{i\in A}) = \emptyset$, or a random sharing $([\boldsymbol{u}], [\boldsymbol{v}]) \in \Sigma(A, (a_i)_{i\in A})$. Note that each share a_i is a pair of elements $(a_i^{(0)}, a_i^{(1)})$ in $\mathbb{K}$.

Depending on the size of A, there are two cases:

- If $|A| \geq t+1$, by the property of the degree-t packed Shamir sharing scheme, the set of shares $\{a_i\}_{i\in A}$ can fully determine the whole sharings if exist. The algorithm parses each share a_i to $(a_i^{(0)}, a_i^{(1)})$. Then, the algorithm checks whether the shares $\{a_i^{(0)}\}_{i\in A}$ lie on a polynomial $f(\cdot)$ of degree at most t in $\mathbb{K}$, and, the shares $\{a_i^{(1)}\}_{i\in A}$ lie on a polynomial $g(\cdot)$ of degree at most t in $\mathbb{K}$. If not, the algorithm outputs $\Sigma(A, (a_i)_{i\in A}) = \emptyset$. Otherwise, the algorithm sets $[\boldsymbol{u}]$ to be the sharing determined by $f(\cdot)$ and $[\boldsymbol{v}]$ to be the sharing determined by $g(\cdot)$. The algorithm checks whether $u_i = v_j$. If not, the algorithm outputs $\Sigma(A, (a_i)_{i\in A}) = \emptyset$. Otherwise, the algorithm outputs $([\boldsymbol{u}], [\boldsymbol{v}])$.

- If $|A| \leq t$, we show that $\Sigma(A, (a_i)_{i\in A}) \neq \emptyset$. The algorithm parses each share a_i to $(a_i^{(0)}, a_i^{(1)})$. To determine the whole sharings, we need additional $t+1-|A|$ shares. For the first sharing, the algorithm randomly samples $t+1-|A|$ elements as the shares of the first $t+1-|A|$ parties in $\mathcal{I}\backslash A$. Then, the first sharing $[\boldsymbol{u}]$ is determined by the shares of the first $t+1-|A|$ parties in $\mathcal{I}\backslash A$ and the shares of parties in A. The algorithm reconstructs the secrets $\boldsymbol{u}$. Note that it fixes one secret in the second sharing, i.e., $v_j = u_i$. For the second sharing, the algorithm randomly samples $t-|A|$ elements as the shares of the first $t-|A|$ parties in $\mathcal{I}\backslash A$. Then, the second sharing $[\boldsymbol{v}]$ is determined by the j-th secret $v_j = u_i$, the shares of the first $t-|A|$ parties in $\mathcal{I}\backslash A$ and the shares of parties in A. Note that $([\boldsymbol{u}], [\boldsymbol{v}])$ is a random sharing in $\Sigma(A, (a_i)_{i\in A})$. The algorithm outputs $([\boldsymbol{u}], [\boldsymbol{v}])$.

Thus, we can use $\mathcal{F}_{\text{rand}}$ with the secret sharing scheme Σ to prepare random sharings $([\boldsymbol{u}], [\boldsymbol{v}])$ such that $u_i = v_j$. Note that the share size of a sharing in Σ is $\mathsf{sh} = 2$ elements in $\mathbb{K}$. Therefore, the communication complexity of preparing m random sharings in Σ is $O(m \cdot n + n^3 \cdot \kappa)$ elements in $\mathbb{K}$.

C Achieving Sublinear Communication Complexity in the Number of Parties

In this part, we explain how to use the techniques in [GIOZ17] to reduce the communication complexity of our main protocol. At a high-level, this is achieved by choosing a small committee of size $O(\log^{1+\delta} n)$ for some constant $\delta > 0$, where n is the number of parties, and evaluating the circuit among parties in this small committee. In more details, suppose the number of corrupted parties is bounded by $t' < (\frac{1}{2} - \epsilon)n$. With all but a negligible probability (in n), the ratio of corrupted parties in the chosen committee is bounded by $\frac{1}{2} - \frac{1}{2}\epsilon$. This is sufficient to use our main protocol.

The solution of choosing a committee in [GIOZ17] works as follows (with a few modifications to fit our setting). For simplicity, we assume the existence of a broadcast channel. We will show how to remove this assumption later.

1. Each party independently decides with probability $p = \frac{\log^{1+\delta} n}{n}$ to volunteer to be a member in the committee. A party that decides to join the committee notifies all other parties by broadcasting its choice. (Note that a party that does not join the committee does nothing.)

2. To prevent too many corrupted parties who volunteer to be in the committee, all partie set $q = (1 + \frac{1}{2}\epsilon) \log^{1+\delta} n$ as the upper bound of the number of parties in the committee. If more than q parties volunteer to be in the committee, all parties abort. We will prove that the threshold q satisfies that:

- If all parties behave honestly, then with all but a negligible probability (in n), the number of parties in the committee is no more than q.
- With all but a negligible probability (in n), the number of honest parties in the committee is at least $(\frac{1}{2}+\frac{1}{2}\epsilon)q$.

Note that with overwhelming probability, the ratio of corrupted parties in the chosen committee is bounded by $(\frac{1}{2}-\frac{1}{2}\epsilon)$.

We will make use of the Chernoff bound stated in Lemma 29.

Lemma 29. *Let $X_1, X_2, \ldots, X_n$ be identical independent random variables taking values in $\{0,1\}$ and p denote the probability that $X_i = 1$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X]$. Then for any $0 \le \eta \le 1$,*

$$\begin{aligned}\Pr[X \le (1-\eta)\cdot\mu] &\le e^{-\frac{1}{2}\eta^2\cdot\mu}\\ \Pr[X \ge (1+\eta)\cdot\mu] &\le e^{-\frac{1}{3}\eta^2\cdot\mu}\end{aligned}$$

Let X_i be the random variable representing whether the i-th party P_i volunteers to be in the committee for all $i \in \{1,2,\ldots,n\}$, and $p = \frac{\log^{1+\delta} n}{n}$.

- If all parties behave honestly, then $X_1, X_2, \ldots, X_n$ are identical independent random variables taking values in $\{0,1\}$ with probability p to be 1. The summation $X = \sum_{i=1}^{n} X_i$ is the number of parties that volunteer to be in the committee. We have $\mu = E[X] = n \cdot p = \log^{1+\delta} n$. Let $\eta = \frac{1}{2}\epsilon$. Then $q = (1+\frac{1}{2}\epsilon)\log^{1+\delta} n = (1+\eta)\cdot\mu$. According to the Chernoff bound, we have

  $$\Pr[X \ge q] = \Pr[X \ge (1+\eta)\cdot\mu] \le e^{-\frac{1}{3}\eta^2\cdot\mu} = e^{-\frac{1}{12}\epsilon^2\cdot\log^{1+\delta} n},$$

  which is negligible in n. Thus, the probability that the number of parties in the committee exceeds q is negligible in n.

- Let $\mathcal{H}$ denote the set of honest parties, then $\{X_i\}_{i\in\mathcal{H}}$ are identical independent random variables taking values in $\{0,1\}$ with probability p to be 1. Let $X' = \sum_{i\in\mathcal{H}} X_i$. Then $\mu' = E[X']$ is the number of honest parties in the committee. Note that there are at least $(\frac{1}{2}+\epsilon)n$ honest parties. Therefore, $\mu' \ge (\frac{1}{2}+\epsilon)\log^{1+\delta} n$. Our goal is to prove that there are at least $(\frac{1}{2}+\frac{1}{2}\epsilon)q = (\frac{1}{2}+\frac{1}{2}\epsilon)(1+\frac{1}{2}\epsilon)\log^{1+\delta} n$ honest parties in the committee. Let

  $$\eta' = 1 - \frac{(\frac{1}{2}+\epsilon)\log^{1+\delta} n}{(\frac{1}{2}+\frac{1}{2}\epsilon)(1+\frac{1}{2}\epsilon)\log^{1+\delta} n} = \frac{\epsilon-\epsilon^2}{2+3\epsilon+\epsilon^2},$$

  which is a constant in $(0,1)$. According to the Chernoff bound, we have

  $$\begin{aligned}\Pr[X' \le (\frac{1}{2}+\frac{1}{2}\epsilon)q] &= \Pr[X' \le (1-\eta')(\frac{1}{2}+\epsilon)\log^{1+\delta} n]\\ &\le \Pr[X' \le (1-\eta')\mu']\\ &\le e^{-\frac{1}{2}(\eta')^2\cdot\mu'}\\ &\le e^{-\frac{1}{2}(\eta')^2\cdot(\frac{1}{2}+\epsilon)\cdot\log^{1+\delta} n},\end{aligned}$$

  which is negligible in n. Therefore, with all but a negligible probability in n, the number of honest parties in the committee is at least $(\frac{1}{2}+\frac{1}{2}\epsilon)q$.

Now we discuss how to achieve broadcasting (with abort) using point-to-point channels:

1. Each party that decides to join the committee sends its choice to all parties using point-to-point channel.

2. Each party locally checks whether the notifications are no more than q and aborts if not. Each party set the committee to be the parties that send notifications. Note that different parties may have different versions of the committee due to the malicious behaviors of corrupted parties in Step 1.

3. Reaching an Agreement on the Committee Members:

   (a) Each party that decides to join the committee sends its own version of the committee members to all other committee members.

   (b) Each party that decides to join the committee checks that whether the lists of committee members it received from other parties are the same as its own version. If not, this party aborts.

4. Notifying the Clients:

   (a) All parties in the committee send their lists of committee members to each client.

   (b) Each client checks whether the lists of committee members it received are the same. If not, this client aborts.

We argue that at the end of the above process, either all honest parties that decide to join the committee and all honest clients reach an agreement on the committee members, or at least one of them aborts. Suppose P_i, P_j are two honest parties that decide to join the committee. In Step 1, they notify each other that they volunteer to be in the committee. Therefore, P_j (or P_i) is in P_i's (or P_j's) list of committee members. In Step 3, they send their own versions of the committee members to each other. Both P_i, P_j abort if the lists are not the same. If no party aborts in Step 3, it means that the lists of P_i, P_j are the same. The same argument applies to each pair of honest parties that decide to join the committee. Therefore, all honest parties that decide to join the committee reach an agreement on the committee members. In Step 4, an honest client either accepts the list received from an honest party or aborts. Therefore all honest clients also agree on the same committee members as honest parties. The statement holds. Note that the (expected) communication complexity of the above process is

$$O(q \cdot n \log n + q^3 \cdot \log n + c \cdot q^2 \cdot \log n) = O((c+n) \cdot \text{poly}(\log n)).$$

Then we may use our maliciously secure protocol among the clients and parties in the chosen committee. The failure probability of our protocol is proportional to $|C|/2^\kappa$, where κ is the security parameter. See Lemma 21, Lemma 22, Lemma 23, and Lemma 25 for more details. To achieve negligible probability in n, we may set $\kappa = \text{poly}(\log n)$. Together with Theorem 8, we have the following theorem:

Theorem 9. *In the client-server model, let c denote the number of clients, and n denote the number of parties (servers). Let $\mathbb{F}$ denote a finite field. For an arithmetic circuit C over $\mathbb{F}$ and any given constant $\epsilon \in (0, \frac{1}{2})$, there exists an information-theoremtic MPC protocol which securely computes the arithmetic circuit C in the presence of a fully malicious adversary controlling up to c clients and $(\frac{1}{2} - \epsilon)n$ parties with all but a negligible probability in n. The communication complexity of this protocol is $O(|C| + (c + n + \mathsf{Depth}) \cdot \mathit{poly}(\log n))$ elements in $\mathbb{F}$. Furthermore, if there exists a broadcast channel, the communication complexity can be reduced to $O(|C| + (c + \mathsf{Depth}) \cdot \mathit{poly}(\log n))$ elements in $\mathbb{F}$ plus $O(\mathit{poly}(\log n))$ bits of broadcasting.*