Perfectly-Secure MPC with Constant Online Communication Complexity

Yifan Song[1] and Xiaxi Ye[2]

[1] Tsinghua University and Shanghai Qi Zhi Institute
yfsong@mail.tsinghua.edu.cn
[2] Tsinghua University
yexx23@mails.tsinghua.edu.cn

Abstract. In this work, we study the communication complexity of perfectly secure MPC protocol with guaranteed output delivery against $t = (n-1)/3$ corruptions. The previously best-known result in this setting is due to Goyal, Liu, and Song (CRYPTO, 2019) which achieves $O(n)$ communication per gate, where n is the number of parties.

On the other hand, in the honest majority setting, a recent trend in designing efficient MPC protocol is to rely on packed Shamir sharings to speed up the online phase. In particular, the work by Escudero et al. (CCS 2022) gives the first semi-honest protocol that achieves a constant communication overhead per gate across all parties in the online phase while maintaining overall $O(n)$ communication per gate. We thus ask the following question: "Is it possible to construct a perfectly secure MPC protocol with GOD such that the online communication per gate is $O(1)$ while maintaining overall $O(n)$ communication per gate?"

In this work, we give an affirmative answer by providing an MPC protocol for computing an arithmetic circuit C over a finite field of size at least $2n$ with communication complexity $O(|C|+\mathsf{Depth}\cdot n+n^5\cdot\log n)$ elements for the online phase, and $O(|C|\cdot n+\mathsf{Depth}\cdot n^2+n^5\cdot\log n)$ elements for the preprocessing phase, where $|C|$ is the circuit size and Depth is the circuit depth.

1 Introduction

Secure multiparty computation (MPC) allows a set of mutually distrustful parties to jointly compute a common function on their private inputs. Very informally, the protocol guarantees that each party can only learn his own input and output but nothing else. Since the notion of MPC was introduced by Yao [Yao82], early feasibility results on MPC were obtained by Yao [Yao82] and Goldreich et al. [GMW87] in the computational setting, where the adversary is assumed to have bounded computational resources. Subsequent works [BGW88,CCD88] considered the unconditional (or information-theoretic) setting and showed positive results up to $t < n/3$ corrupted parties assuming point-to-point communication channels. If one assumes a broadcast channel in addition, it was shown in [RB89,Bea89] how to obtain positive results in the information-theoretic setting for up to $t < n/2$ corrupted parties.

In this work, we are interested in the communication complexity of perfectly secure MPC with guaranteed output delivery (GOD) over point-to-point channels. At a high level, perfect security requires that any (computationally unbounded) adversary by controlling t corrupted parties cannot learn any information about honest parties' inputs even if corrupted parties can arbitrarily deviate from the protocol. Guaranteed output delivery, on the other hand, requires that the protocol should always succeed. Indeed, perfect security with GOD is the best possible security one can hope. It has been shown in [BGW88] that perfectly secure MPC with GOD is impossible to achieve when $t \geq n/3$. On the other hand, when $t < n/3$, [BGW88] gives the first positive result that can compute any computable functions.

Communication complexity is an important measurement of the efficiency of an MPC protocol, especially in the information-theoretic setting. This is because unconditionally secure MPC protocols usually have light weight local computation, often just a series of linear operations. On the other hand, known constructions for general functions still require at least a linear communication complexity in the circuit size. Thus in the real world, the efficiency of an unconditional MPC protocol is dominated by its communication complexity.

There is a rich line of works studying the communication complexity of perfectly secure MPC with GOD. Most noticeably, the work [BH08] gives the first result with linear communication complexity per multiplication gate. To be more concrete, the achieved communication complexity is $O(|C| \cdot n + \mathsf{Depth} \cdot n^2 + n^3)$ elements. In 2019, Goyal et al. [GLS19] show how to remove the quadratic communication overhead in the circuit depth and achieve $O(|C| \cdot n + n^3)$ elements. Both of these works are based on the party elimination framework [HMP00], a generic approach to achieve GOD efficiently. On the other hand, this approach inherently requires $O(n)$ evaluation rounds. Therefore, the round complexity of these two works are $O(\mathsf{Depth} + n)$. Another line of works [ALR11,AAY22,AAPP23] focuses on improving the communication complexity without the $O(n)$ overhead in the round complexity. And in the recent work [AAPP23], a linear communication overhead per gate in the number of parties is also achieved in this setting. Concretely the achieved communication complexity is $O(|C| \cdot n + \mathsf{Depth} \cdot n^2 + n^4)$. It is not clear whether a linear communication overhead per gate is inherent in general in perfect malicious security setting despite there are negative results for special cases with an additional requirement that the protocol be two-phase [DS20]. [1]

We note that in the honest majority setting where $t < n/2$, the communication complexity of the best-known semi-honest protocol [DN07,GLO+21] is also linear in the number of parties per gate. However, a recent work [EGPS22] gives a novel construction that achieves $O(1)$ communication per gate in the online phase while preserving $O(n)$ overall communication per gate in the preprocessing phase. Having $O(1)$ communication per gate in the online phase is interesting since

- In practice, people care more about the efficiency in the online phase as the preprocessing phase can be done in the idle time before knowing the inputs.
- It means that the amortized communication complexity per party decreases as the increase of the number of parties and one may speed up the protocol by having more parties![1]

Unfortunately, to the best of our knowledge, such a result is not known in the perfect security setting. Thus, we ask the following question:

"Is it possible to construct a perfectly secure MPC protocol with GOD such that the online communication complexity per gate is $O(1)$ while the overall communication remains $O(n)$?"

1.1 Our Contribution

In this work, we answer the above question affirmatively. Our main result is summarized in the following theorem.

Theorem 1. *Let n denote the number of parties. Let $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq 2n$. For an arithmetic circuit C over $\mathbb{F}$, there exists an information-theoretic MPC protocol that computes C against a fully malicious adversary controlling at most $t = \frac{n-1}{3}$ corrupted parties with perfect security. The communication cost of the protocol is expected $O(|C| + \mathsf{Depth} \cdot n + n^5 \cdot \log n)$ elements for the online phase and expected $O(|C| \cdot n + \mathsf{Depth} \cdot n^2 + n^5 \cdot \log n)$ elements for the preprocessing phase, where Depth is the circuit depth. The round complexity of the protocol is $O(\mathsf{Depth} + n^2)$ in expectation for the online phase and $O(n^2)$ in expectation for the preprocessing phase.*

We note that the previously best-known results [GLS19,AAPP23] both require linear communication complexity per gate in the online phase. Thus, our result gives a factor of $O(n)$ improvements over the previous results in the online phase.

To achieve our result, our idea is to compile the semi-honest protocol in [EGPS22] to achieve perfect security. While the semi-honest protocol in [EGPS22] gives us an efficient way to compute the circuit in the online phase, we note that the main difficulty of achieving perfect security is to efficiently verify the

[1] The negative result in [DS20] requires that protocols are UC secure and have a standard 2-phase structure where the inputs are committed in the first phase, before the output is computed in the second phase.

[1] Our construction requires the field size to be $O(n)$. We leave the extension of our result to constant-size fields to future work.

computation and locate the errors. In particular, we identify a security issue that does not occur in the semi-honest setting when compiling [EGPS22] to achieve perfect security. We note that this security issue is very similar to the one observed in [GLS19]. Compared with [GLS19], we give a much simpler solution to address this issue which can potentially be used to improve the construction in [GLS19]. In Section 2, we give an overview of our solution towards tackling these difficulties.

Comparison with Previous Works. As we have mentioned above, our work achieves the same overall asymptotic communication complexity as [GLS19,AAPP23] while we achieve constant online communication per gate.

On the other hand,

- Both of our work and [AAPP23] suffer an additive communication overhead depending on the circuit depth, $O(\mathsf{Depth} \cdot n^2)$, while [GLS19] does not have this term.
- For round complexity, when using an expected constant round BC protocol, our protocol requires $O(\mathsf{Depth} + n^2)$ rounds in expectation due to the use of the dispute control framework. This is in contrast to [AAPP23] that achieves expected $O(\mathsf{Depth})$ rounds and [GLS19] that achieves expected $O(\mathsf{Depth} + n)$ rounds.

This additional communication overhead depending on the circuit depth and additional round complexity may be viewed as the tradeoff of achieving constant online communication per gate. An interesting future direction is to achieve the best among these three works.

When focusing on the non-optimal corruption setting, [DIK10] achieves sublinear communication and computation in the number of parties. As for its techniques, [DIK10] first transforms a general circuit into a SIMD-like circuit to overcome the issue of network routing where the input wires for each group of gates need to be aligned. However, this transformation increases the circuit size by a factor of $\log |C|$. To evaluate the circuit after transformation, [DIK10] designs a protocol for a much smaller corruption threshold and use the party virtualization technique to boost the corruption threshold. However, the technique of party virtualization only works in the non-optimal corruption setting which does not work for the optimal 1/3 corruption setting we target for. We note that if only focusing on the communication complexity, one can potentially achieve overall $O(|C|)$ communication by adapting the protocol in [GPS21] that addresses network routing with constant overhead and combining it with our verification technique. However, one main challenge in our case is that, to achieve $O(|C|)$ online communication, we have to use packed Shamir sharing with larger degree $d(> n/3)$ to pack $O(n)$ secrets which leads to the loss of error correction property. In contrast, both [DIK10] and the above adaption of [GPS21] can use Shamir sharings with degree $d = n/3 - 1$ while still packing $O(n)$ secrets and take advantage of the error correction property to achieve a much simpler protocol.

2 Technical Overview

Our work uses both the standard Shamir sharings and Packed Shamir sharings. We use $[x]_d$ to represent a degree-d Shamir sharing of x, which corresponds to a degree-d polynomial f such that the i-th share is $f(i)$, and $f(0) = x$. We will also use $[x|j]_d$ to denote a degree-d Shamir sharing of x where the secret value is stored at $f(-j+1)$ rather than $f(0)$. For a vector $\boldsymbol{x} = (x_1, \ldots, x_k) \in \mathbb{F}^k$, we use $[\boldsymbol{x}]_d$ to represent a degree-d packed Shamir sharing of $\boldsymbol{x}$, which corresponds to a degree-d polynomial f such that the i-th share is $f(i)$ and for all $j \in [k]$, $f(-j+1) = x_j$.

2.1 Efficient Online Protocol via Preprocessing

In this work, our goal is to design a perfectly secure MPC protocol with guaranteed output delivery (GOD) against a fully malicious adversary who controls up to $t = \frac{n-1}{3}$ corrupted parties such that the online communication complexity per multiplication gate is constant among all parties. Our starting point is the

recent semi-honest protocol [EGPS22] in the honest majority setting that achieves constant communication complexity in the online phase relying on the packed Shamir sharing scheme and preprocessing.

The notion of packed Shamir sharing was introduced by Franklin and Yung in [FY92]. At a high level, packed Shamir sharings allow us to compute a batch of $O(n)$ gates of the same kind simultaneously but only at cost $O(n)$, the same as using the Shamir sharings to compute a single gate. This allows us to bring the cost per gate to $O(1)$. In [EGPS22], the authors rely on preprocessing to prepare packed Beaver triples (the packed version of the standard Beaver triples) which are used in the online phase to achieve constant online communication. On the other hand, the overall communication complexity of the construction in [EGPS22] remains $O(n)$ per gate, which matches the best-known semi-honest protocol [DN07,GLO+21].

Inspired by [EGPS22], our idea is to rely on packed Shamir sharings to achieve perfect security with GOD with constant online communication per gate. At a very high level, our idea is to (1) use techniques in [BH08] to compile the preprocessing phase of [EGPS22] to prepare packed Beaver triples, and (2) use party elimination framework [HMP00] to compile the online protocol that uses packed Beaver triples. However, the second step is much more difficult to achieve than it looks. In the following, we give an overview of our construction and demonstrate the technical difficulties we have to address. For simplicity, we first focus on a SIMD circuit, which computes many copies of the same sub-circuit. We will discuss how to move to a general circuit later.

Overview of Our Protocol. Let k be the number of secrets packed in a single sharing. We will discuss the choice of k at a later point. We set $d = t + k - 1$ and use degree-d packed Shamir sharings to ensure privacy against t corrupted parties. A packed Beaver triple contains three degree-d packed Shamir sharings $([\boldsymbol{a}]_d, [\boldsymbol{b}]_d, [\boldsymbol{c}]_d)$ such that $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$, where $*$ denotes the coordinate-wise multiplication.

In the preprocessing phase of [EGPS22], a random packed Beaver triple is prepared as follows.

- First, for all $i \in [k]$, a standard Beaver triple with secrets stored at position $-i+1$ is prepared: $([a_i|i]_t, [b_i|i]_t, [c_i|i]_t)$.
- Then, all parties locally convert such a group of k Beaver triples to a packed Beaver triple. This is done by computing $[\boldsymbol{a}]_d = \sum_{i=1}^{k} [\boldsymbol{e}_i]_{k-1} \cdot [a_i|i]_t$, where $\boldsymbol{e}_i$ is the i-th unit vector (i.e., the i-th entry is 1 while all other entries are 0). To see why it works, just note that the secrets of $[\boldsymbol{e}_i]_{k-1} \cdot [a_i|i]_t$ is equal to $a_i \cdot \boldsymbol{e}_i$. Thus, we have $\boldsymbol{a} = \sum_{i=1}^{k} a_i \cdot \boldsymbol{e}_i$.

Using techniques in [BH08], we can prepare standard Beaver triples with perfect security. Thus, following [EGPS22], we can efficiently prepare packed Beaver triples with perfect security.

With packed Beaver triples in hand, in the online phase, all parties evaluate a group of k gates in parallel each time. For a group of k addition gates with input sharings $[\boldsymbol{x}]_d, [\boldsymbol{y}]_d$, all parties locally compute $[\boldsymbol{z}]_d = [\boldsymbol{x}]_d + [\boldsymbol{y}]_d$. For a group of k multiplication gates, all parties run the following steps:

1. All parties locally compute $[\boldsymbol{x}+\boldsymbol{a}]_d = [\boldsymbol{x}]_d + [\boldsymbol{a}]_d$ and $[\boldsymbol{y}+\boldsymbol{b}]_d = [\boldsymbol{y}]_d + [\boldsymbol{b}]_d$ and send them to a common party P_{king}.
2. P_{king} reconstructs $\boldsymbol{x}+\boldsymbol{a}, \boldsymbol{y}+\boldsymbol{b}$, and distributes $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{k-1}$ to all parties.
3. All parties locally compute

$$[\boldsymbol{z}]_{d+k-1} = [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{b}]_d - [\boldsymbol{a}]_d \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} + [\boldsymbol{c}]_d.$$

   To obtain a degree-d packed Shamir sharing of $\boldsymbol{z}$, all parties in addition prepare $([\boldsymbol{r}]_d, [\boldsymbol{r}]_{d+k-1})$. Such a pair of random sharings can also be prepared by using techniques in [BH08].
4. All parties locally compute $[\boldsymbol{z}+\boldsymbol{r}]_{d+k-1} = [\boldsymbol{z}]_{d+k-1} + [\boldsymbol{r}]_{d+k-1}$ and send it to P_{king}.
5. P_{king} reconstructs $\boldsymbol{z}+\boldsymbol{r}$ and distributes $[\boldsymbol{z}+\boldsymbol{r}]_{k-1}$ to all parties.
6. All parties locally compute $[\boldsymbol{z}]_d = [\boldsymbol{z}+\boldsymbol{r}]_{k-1} - [\boldsymbol{r}]_d$.

In this way, all parties can evaluate every group of gates with constant communication.

When corrupted parties deviate from the protocol, however, the above protocol can easily go wrong. This is because the packed Shamir sharings are of degree d which is greater than t, and we cannot hope to use the

error correction of Reed Solomon Code as [BH08] to ensure the correctness of the computation in the online phase. To achieve perfect security, one may hope to use the party elimination framework as [GLS19]. At a high level, the whole computation task is first divided into $O(n)$ segments. Each time one segment is computed as above. Then all parties together check the correctness of the computation. If the computation is correct, all parties move to the next segment. Otherwise, all parties together find a pair of dispute parties which ensures at least one party is corrupted. Such a dispute pair is removed and all parties re-evaluate the current segment. This way, we can achieve perfect security without blowing up the communication complexity.

Unfortunately, this idea does not work in our case. This is because in the worst case, the party elimination framework may remove $2t$ parties and only $t+1$ parties left. However, in the above multiplication protocol, to allow P_{king} to reconstruct $[\boldsymbol{z}]_{d+k-1}$, at least $d+k = t+2k-1 > t+1$ parties are needed. To resolve this issue, our construction uses the dispute control method [BH06]. The high-level idea of dispute control is similar to party elimination framework except that each time a dispute pair is found, we will not remove these two parties. Instead, we will ensure that these two parties never talk to each other in the following computation. In this way, we could avoid finding the same dispute pair in the future evaluation. A party is only removed if he is disputed with more than t parties, in which case this party is definitely a corrupted party. By using dispute control, we will only remove corrupted parties. Thus, at least $2t+1$ (honest) parties are active. To ensure that P_{king} can always receive enough shares for $[\boldsymbol{z}]_{d+k-1}$, which requires $d+k = t+2k-1$ shares, we need $t+2k-1 \leq 2t+1$. Thus, our construction sets $k = \frac{t}{2}+1 = O(n)$, which is sufficient to achieve our goal.

Note that so far we haven't discussed how to check the correctness of the computation and how to find a dispute pair if the computation goes wrong. In particular,

- The verification of computation should be done with constant communication per gate as well. Note that this difficulty does not appear in [GLS19] since their construction only achieves linear communication per gate in the online phase. So it suffices for them to also pay linear cost per gate in the verification. This difficulty does not appear in [EGPS22] either since their malicious version is in the honest majority setting where one cannot hope to achieve perfect security. However, allowing errors with negligible probability simplifies the task of verification as one can even achieve sublinear cost in the verification [GSZ20].
- Degree-d packed Shamir sharings are insufficient to identify dispute pairs when the computation goes wrong. This is unlike degree-t Shamir sharings where even if all corrupted parties provide incorrect shares, we can always reconstruct the whole sharing. In fact, this is a much more severe issue since if the input packed Shamir sharings are found to be incorrect, then even if all parties follow the protocol, the computation will always fail and we are not able to find a dispute pair.

In the following sections, we will address these two difficulties.

2.2 Boosting Verification

As we discussed above, we have to design a verification protocol with constant communication per gate. We first examine where the multiplication protocol may go wrong.

- **Issue 1**. In Step 1 and 4, parties may send incorrect shares to P_{king} so that P_{king} cannot reconstruct the secrets of degree-d or degree-$(d+k-1)$ packed Shamir sharings.
- **Issue 2**. In Step 2 and 5, P_{king} may maliciously distribute incorrect packed sharings where either the degree of packed sharings is not $k-1$ or the secrets are incorrect.

In the first case, we show that P_{king} can detect such issues and directly announce to others that the computation is incorrect. Recall that we set k, the number of secrets packed in a single sharing, to be $\frac{t}{2}+1$. Then $(d+k-1)+1 = t+2k-1 = 2t+1$. Therefore, a degree-$d$ or degree-$(d+k-1)$ packed Shamir sharing is fully determined by the shares of honest parties and corrupted parties can only cause the sharing to be inconsistent but not change the secret. Thus, P_{king} can detect errors by checking whether the received shares lie on a valid degree-d or degree-$(d+k-1)$ polynomial.

In the second case, we can abstract the verification task as follows. All parties hold a pair of packed Shamir sharings $([\boldsymbol{u}]_{d+k-1}, [\boldsymbol{u}]_{k-1})$, where the first sharing is the one all parties send to P_{king} and the second

sharing is the one all parties receive from P_{king} (note that we can always view a degree-d packed Shamir sharing as a degree-$(d+k-1)$ packed Shamir sharing). The goal is to check that (1) the second sharing is a valid degree-$(k-1)$ packed Shamir sharing, and (2) both sharings have the same secrets. To verify such a pair, we let each party receive the shares from all parties and check the above two points accordingly. Here we rely again on the fact that a degree-$(d+k-1)$ packed Shamir sharing is fully determined by the shares of honest parties. Thus corrupted parties cannot make honest parties accept the verification by sending wrong shares.

However this way of checking $([\boldsymbol{u}]_{d+k-1}, [\boldsymbol{u}]_{k-1})$ would require $O(n^2)$ communication per pair. To amortize the communication complexity, we adapt the technique in [BH08] to efficiently check a batch of $2t+1$ such pairs, say $\{([\boldsymbol{u}_i]_{d+k-1}, [\boldsymbol{u}_i]_{k-1})\}_{i=1}^{2t+1}$. Let $\boldsymbol{M}$ be a Vandermonde matrix of size $n \times (2t+1)$. All parties first expand such $2t+1$ pairs to n pairs by locally computing

$$([\boldsymbol{v}_i]_{d+k-1}, [\boldsymbol{v}_i]_{k-1})_{i=1}^{n} = \boldsymbol{M} \cdot ([\boldsymbol{u}_i]_{d+k-1}, [\boldsymbol{u}_i]_{k-1})_{i=1}^{2t+1}.$$

By the property of Vandermonde matrices, there is a one-to-one linear map between any subset of $2t+1$ pairs in $\{([\boldsymbol{v}_i]_{d+k-1}, [\boldsymbol{v}_i]_{k-1})\}_{i=1}^{n}$ and $\{([\boldsymbol{u}_i]_{d+k-1}, [\boldsymbol{u}_i]_{k-1})\}_{i=1}^{2t+1}$. Thus if $2t+1$ pairs in $\{([\boldsymbol{v}_i]_{d+k-1}, [\boldsymbol{v}_i]_{k-1})\}_{i=1}^{n}$ are correct, this implies that $\{([\boldsymbol{u}_i]_{d+k-1}, [\boldsymbol{u}_i]_{k-1})\}_{i=1}^{2t+1}$ are all correct. Therefore, after expansion, we let each party P_i check a single pair $([\boldsymbol{v}_i]_{d+k-1}, [\boldsymbol{v}_i]_{k-1})$. If every party is happy with the pair he checked, then at least $2t+1$ pairs are verified by honest parties, which ensures that the original $2t+1$ pairs are correct. Finally, all parties run a Byzantine Agreement protocol to reach an agreement on whether the verification passes or not. Note that the communication remains to be $O(n^2)$ but we check $O(n)$ pairs each time.

In Section 5, we extend the above idea to check any linear secret sharing scheme which satisfies that the whole sharing is determined by the shares of honest parties. The functionality $\mathcal{F}_{\mathsf{VerifyPub}}$ and protocol $\Pi_{\mathsf{VerifyPub}}$ will be formally described in Subsection 4.1 and Subsection 5.1, respectively.

2.3 Identifying Dispute Pair

After the verification, if the check fails, we reach a scenario where

- Either P_{king} claims that the degree-d or degree-$(d+k-1)$ packed Shamir sharing he received is incorrect.
- Or some party P_i claims that $([\boldsymbol{v}_i]_{d+k-1}, [\boldsymbol{v}_i]_{k-1})$ he received is incorrect.

Then we know the computation of the current segment fails and we have to identify a dispute pair of parties. For the latter case, since $([\boldsymbol{v}_i]_{d+k-1}, [\boldsymbol{v}_i]_{k-1})$ is computed from $\{([\boldsymbol{u}_i]_{d+k-1}, [\boldsymbol{u}_i]_{k-1})\}_{i=1}^{2t+1}$ which are known by P_{king}, P_{king} can provide a correct version to P_i so that P_i can cross check which party deviates from the protocol. However, what if P_{king} claims that a degree-d or degree-$(d+k-1)$ packed Shamir sharing is incorrect? Since $d > t$, we cannot hope to always reconstruct the correct sharing and identify the party who sends the wrong share.

To enable identification of dispute pair, we have to resort to degree-t Shamir sharings. Our idea is to compute a degree-t Shamir sharing for every value and all parties hold these degree-t Shamir sharings silently. In this way, whenever a degree-d or degree-$(d+k-1)$ packed Shamir sharing is incorrect, we can always come back to the degree-t sharings to identify a dispute pair. However, we have to achieve this with constant online communication.

Computing Degree-t Sharings with Constant Online Communication. Our observation is that in the preprocessing phase of [EGPS22], all parties can obtain a degree-d packed Shamir sharing $[\boldsymbol{a}]_d$ via local computation from $\{[a_i|i]_t\}_{i=1}^{k}$. This inspires us to compute degree-t Shamir sharings in the online phase with a similar form so that when we compute multiplication gates, we can pack the degree-t Shamir sharings on demand. To be more concrete, for every group of k multiplication gates, all parties hold $\{[x_i|i]_t, [y_i|i]_t\}_{i=1}^{k}$ in the beginning. They first locally transform these $2k$ degree-t Shamir sharings to $[\boldsymbol{x}]_d, [\boldsymbol{y}]_d$ and then run the multiplication protocol.

Note that we also have to unpack the output sharing $[\boldsymbol{z}]_d$ to $\{[z_i|i]_t\}_{i=1}^{k}$. Fortunately, this can be achieved with a small modification of the preprocessing data: We require all parties to also prepare $\{[r_i|i]_t\}_{i=1}^{k}$ when

preparing $([\boldsymbol{r}]_d, [\boldsymbol{r}]_{d+k-1})$. Then after receiving $[\boldsymbol{z}+\boldsymbol{r}]_{k-1}$ from P_{king}, since $[\boldsymbol{z}+\boldsymbol{r}]_{k-1}$ can be viewed as a degree-$(k-1)$ Shamir sharing of $z_i + r_i$ stored at $-i+1$, i.e., $[z_i + r_i|i]_{k-1}$, all parties can compute $[z_i|i]_t = [\boldsymbol{z}+\boldsymbol{r}]_{k-1} - [r_i|i]_t$.

Identifying Dispute Pair When $[\boldsymbol{x}+\boldsymbol{a}]_d$ is Incorrect. Now come back to the problem of identifying dispute pair. If P_{king} claims that $[\boldsymbol{x}+\boldsymbol{a}]_d$ is incorrect, all parties can provide $\{[x_i + a_i|i]_t\}_{i=1}^k$ (note that $[a_i|i]_t$ is generated when preparing packed Beaver triples in the preprocessing phase). In this way, P_{king} can robustly reconstruct each degree-t Shamir sharing and compute the correct $[\boldsymbol{x}+\boldsymbol{a}]_d$.

Identifying Dispute Pair When $[\boldsymbol{z}+\boldsymbol{r}]_{d+k-1}$ is Incorrect. If P_{king} claims that $[\boldsymbol{z}+\boldsymbol{r}]_{d+k-1}$ is incorrect, however, the above approach does not work. Recall that

$$[\boldsymbol{z}+\boldsymbol{r}]_{d+k-1} = [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{b}]_d - [\boldsymbol{a}]_d \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} + [\boldsymbol{c}]_d + [\boldsymbol{r}]_{d+k-1},$$

where $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{k-1}$ are distributed by P_{king}. Also recall that all parties have prepared $\{([a_i|i]_t, [b_i|i]_t, [c_i|i]_t)\}_{i=1}^k$ in the preprocessing phase. To allow P_{king} robustly reconstructing $[\boldsymbol{z}+\boldsymbol{r}]_{d+k-1}$, we further change the way of preparing $[\boldsymbol{r}]_{d+k-1}$ as follows: All parties prepare $2k-1$ random degree-t Shamir sharings $\{[r_i|i]_t\}_{i=1}^{2k-1}$. Let $\boldsymbol{r}' = (r_1, \ldots, r_{2k-1})$. Following the observation in [EGPS22], all parties can locally transform $\{[r_i|i]_t\}_{i=1}^{2k-1}$ to a degree-$(d+k-1)$ packed Shamir sharing $[\boldsymbol{r}']_{d+k-1}$ (that stores $2k-1$ secrets). Here we utilize the fact that $d+k-1 = t+(2k-1)-1$ by our choices of d and k. Note that if we only focus on the first k secrets of $\boldsymbol{r}'$, denoted by $\boldsymbol{r} = (r_1, \ldots, r_k)$, $[\boldsymbol{r}']_{t+2k-2}$ can be directly viewed as a degree-$(d+k-1)$ packed Shamir sharing $[\boldsymbol{r}]_{d+k-1}$ (that stores k secrets), which is what we need.

Now if all parties send $\{([a_i|i]_t, [b_i|i]_t, [c_i|i]_t)\}_{i=1}^k$ and $\{[r_i|i]_t\}_{i=1}^{2k-1}$ to P_{king}, P_{king} can reconstruct each degree-t Shamir sharing and compute a correct degree-$(d+k-1)$ packed Shamir sharing $[\boldsymbol{z}+\boldsymbol{r}]_{d+k-1}$, and therefore can identify a dispute pair. However, doing this would reveal $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ to P_{king}. To resolve this issue, we use the following tricks.

1. In the preprocessing phase, all parties prepare random degree-t Shamir sharings $\{([a'_i|i]_t, [b'_i|i]_t, [c'_i|i]_t)\}_{i=1}^k$ and $\{[r'_i|i]_t\}_{i=1}^{2k-1}$ as random masks. Note that each c'_i is a random value rather than $a'_i \cdot b'_i$.
2. All parties compute $[\boldsymbol{a}']_d, [\boldsymbol{b}']_d, [\boldsymbol{c}']_d, [\boldsymbol{r}']_{2k-1}$ locally. Then all parties send

   $$[\boldsymbol{v}]_{d+k-1} = [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{b}']_d - [\boldsymbol{a}']_d \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} + [\boldsymbol{c}']_d + [\boldsymbol{r}']_{d+k-1}$$

   to P_{king}. Basically, $[\boldsymbol{v}]_{d+k-1}$ is computed in the same way as $[\boldsymbol{z}+\boldsymbol{r}]_{d+k-1}$ except that we use $[\boldsymbol{a}']_d, [\boldsymbol{b}']_d, [\boldsymbol{c}']_d, [\boldsymbol{r}']_{d+k-1}$.
3. Now if P_{king} complains about $[\boldsymbol{v}]_{d+k-1}$, all parties send $\{([a'_i|i]_t, [b'_i|i]_t, [c'_i|i]_t)\}_{i=1}^k$ and $\{[r'_i|i]_t\}_{i=1}^{2k-1}$ to P_{king}. These are independent of the actual wire values.
4. Otherwise, it means that $[\boldsymbol{z}+\boldsymbol{r}]_{d+k-1} + [\boldsymbol{v}]_{d+k-1}$ is not a valid degree-$(d+k-1)$ packed Shamir sharing. Note that we have

   $$\begin{aligned}&[\boldsymbol{z}+\boldsymbol{r}]_{d+k-1} + [\boldsymbol{v}]_{d+k-1}\\ =\ &2[\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot ([\boldsymbol{b}]_d + [\boldsymbol{b}']_d) - ([\boldsymbol{a}]_d + [\boldsymbol{a}']_d) \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1}\\ &+([\boldsymbol{c}]_d + [\boldsymbol{c}']_d) + ([\boldsymbol{r}]_{d+k-1} + [\boldsymbol{r}']_{d+k-1}).\end{aligned}$$

   All parties send $\{([a_i+a'_i|i]_t, [b_i+b'_i|i]_t, [c_i+c'_i|i]_t)\}_{i=1}^k$ and $\{[r_i+r'_i|i]_t\}_{i=1}^{2k-1}$ to P_{king}. Again those sharings are independent of the actual wire values because of the random masks.

In this way, P_{king} can identify a party who sends incorrect shares.

2.4 Security Issue of the Current Approach

One issue we omitted so far is that the multiplication above is actually not secure against malicious corrupted parties: A malicious P_{king} can learn some partial information of $\boldsymbol{y}$ from the shares of $[\boldsymbol{z}+\boldsymbol{r}]_{d+k-1}$ by sending incorrect $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{k-1}$. Consider the following so-called double-dipping attack [GBO+23,DEN24] which has been pointed out in [GLS19]:

1. Without loss of generality, assume the first $2t+1$ parties are honest. After P_{king} reconstructs $\boldsymbol{e} = \boldsymbol{x} + \boldsymbol{a}$, P_{king} chooses another vector $\boldsymbol{e}'$ such that the first shares of $[\boldsymbol{e}]_{k-1}$ and $[\boldsymbol{e}']_{k-1}$ are identical. Then P_{king} sends the shares of $[\boldsymbol{e}]_{k-1}$ to $\{P_1, \ldots, P_{t+1}\}$, and sends the shares of $[\boldsymbol{e}']_{k-1}$ to $\{P_{t+2}, \ldots, P_{2t+1}\}$.
2. Let $[\boldsymbol{z} + \boldsymbol{r}]_{d+k-1}$ denote the correct sharings computed from $[\boldsymbol{e}]_{k-1}$ and $[\boldsymbol{z}' + \boldsymbol{r}]_{d+k-1}$ denote the incorrect sharings computed from $[\boldsymbol{e}']_{k-1}$. Then the parties in $\{P_1, \ldots, P_{t+1}\}$ hold shares of $[\boldsymbol{z} + \boldsymbol{r}]_{d+k-1}$ while parties in $\{P_1, P_{t+2}, \ldots, P_{2t+1}\}$ hold shares of $[\boldsymbol{z}' + \boldsymbol{r}]_{d+k-1}$. Note that corrupted parties, i.e., $\{P_{2t+2}, \ldots, P_{3t+1}\}$, can compute their shares of both $[\boldsymbol{z} + \boldsymbol{r}]_{d+k-1}$ and $[\boldsymbol{z}' + \boldsymbol{r}]_{d+k-1}$.
3. After receiving the shares of $[\boldsymbol{z} + \boldsymbol{r}]_{d+k-1}$ (or $[\boldsymbol{z}' + \boldsymbol{r}]_{d+k-1}$) from all honest parties, P_{king} learns $2t+1$ shares of both $[\boldsymbol{z} + \boldsymbol{r}]_{d+k-1}$ and $[\boldsymbol{z}' + \boldsymbol{r}]_{d+k-1}$. Thus P_{king} can reconstruct $\boldsymbol{z} + \boldsymbol{r}$ and $\boldsymbol{z}' + \boldsymbol{r}$ and compute $\boldsymbol{z}' - \boldsymbol{z} = (\boldsymbol{e}' - \boldsymbol{e}) * \boldsymbol{y}$, which leaks the information about $\boldsymbol{y}$.

We point out that such an issue does not appear in [EGPS22] since in their setting, they set $d+k-1 = n-1$ so that P_{king} needs all shares to reconstruct $\boldsymbol{z} + \boldsymbol{r}$. In our case, however, we need $d + k - 1 \leq 2t$ to ensure that P_{king} can detect the errors in $[\boldsymbol{z} + \boldsymbol{r}]_{d+k-1}$.

We note that a similar issue has been pointed out in [GLS19]. We follow the approach in [GLS19] to resolve this issue. Before sending $[\boldsymbol{z} + \boldsymbol{r}]_{d+k-1}$ to P_{king}, all parties prepare a random degree-$(n-1)$ packed Shamir sharing of $\mathbf{0}$, denoted by $[\mathbf{0}]_{n-1}$, and add it with $[\boldsymbol{z} + \boldsymbol{r}]_{d+k-1}$. In this way, even if a malicious P_{king} may distribute incorrect $[\boldsymbol{x} + \boldsymbol{a}]_{k-1}$ and $[\boldsymbol{y} + \boldsymbol{b}]_{k-1}$, he no longer gains any information from shares of $[\boldsymbol{z} + \boldsymbol{r}]_{n-1}$. Intuitively, this is because $[\boldsymbol{z} + \boldsymbol{r}]_{n-1} = [\boldsymbol{z}]_{d+k-1} + ([\boldsymbol{r}]_{d+k-1} + [\mathbf{0}]_{n-1})$ where the second part is a random degree-$(n-1)$ packed Shamir sharing and every share is just a random value. Effectively, every party uses a uniform value to hide his share of $[\boldsymbol{z}]_{d+k-1}$.

While this approach prevents a corrupted P_{king} from gaining information from $[\boldsymbol{z} + \boldsymbol{r}]_{n-1}$, an honest P_{king} cannot detect errors in $[\boldsymbol{z} + \boldsymbol{r}]_{n-1}$ either. In [GLS19], the authors introduce the so called 4-consistent sharings to detect errors in $[\boldsymbol{z} + \boldsymbol{r}]_{n-1}$. In our work, we give a much simpler approach for this task, which can also be used to simplify the construction in [GLS19] and avoid the use of 4-consistent sharings.

We notice that the key point of the above attack is that P_{king} may distribute $[\boldsymbol{x} + \boldsymbol{a}]_{k-1}$ and $[\boldsymbol{y} + \boldsymbol{b}]_{k-1}$ that are not of degree $(k-1)$. On the other hand, if P_{king} is guaranteed to distribute degree-$(k-1)$ packed Shamir sharings, even if the secrets are incorrect, P_{king} cannot learn any information from $[\boldsymbol{z} + \boldsymbol{r}]_{d+k-1}$ since in this case it is a valid degree-$(d+k-1)$ packed Shamir sharing and the potentially incorrect secrets $\boldsymbol{z}$ are masked by $\boldsymbol{r}$. Thus, to allow P_{king} to detect errors in $[\boldsymbol{z} + \boldsymbol{r}]_{n-1}$,

1. All parties first check that P_{king} indeed shares degree-$(k-1)$ packed Shamir sharings. This check can be done by the verification protocol $\Pi_{\mathsf{VerifyPub}}$ we introduced above.
2. All parties together open the mask sharing $[\mathbf{0}]_{n-1}$. This is done by letting each party send all messages when generating $[\mathbf{0}]_{n-1}$ to P_{king}. Note that given that P_{king} shares valid degree-$(k-1)$ packed Shamir sharings, $[\mathbf{0}]_{n-1}$ is safe to open. Now P_{king} can detect errors in $[\boldsymbol{z} + \boldsymbol{r}]_{d+k-1} = [\boldsymbol{z} + \boldsymbol{r}]_{n-1} - [\mathbf{0}]_{n-1}$ again.

In our construction, we will also use a random packed Shamir sharing of $\mathbf{0}$ when reconstructing $[\boldsymbol{x} + \boldsymbol{a}]_d$ and $[\boldsymbol{y} + \boldsymbol{b}]_d$ to P_{king}. In this way, we can first evaluate multiplication gates in multiple layers without check, and then verify the correctness of multiplication gates at the end. We point out that without doing this, a similar security issue occurs when evaluating multiplication gates in multiple layers without check [GLS19].

To summarize, we use techniques in [GLS19] to allow computation across layers and only verify the computation at the end of each segment. To achieve this, we add a degree-$(n-1)$ packed Shamir sharings of $\mathbf{0}$ to remove the redundancy of the sharings reconstructed to P_{king} in order to prevent P_{king} from applying the double-dipping attack by distributing inconsistent reconstruction results to all parties. However, when the computation fails, P_{king} cannot pin-point which party misbehaved due to the lack of redundancy. Our observation is that the high-degree packed Shamir sharings of $\mathbf{0}$ is to prevent P_{king} from applying the double-dipping attack and once this has been checked (i.e., P_{king} indeed sends consistent reconstruction results to all parties), it is safe for all parties to de-mask the random sharings of $\mathbf{0}$. Hence, P_{king} could again rely on the redundancy of lower degree Shamir sharings for verification.

Using Our Technique in [GLS19]. The construction in [GLS19] uses the standard degree-t Shamir sharing and random Beaver triples for online computation. To compute the multiplication of two input sharings

$[x]_t, [y]_t$ with Beaver triple $([a]_t, [b]_t, [c]_t)$, all parties first reconstruct $x+a$ and $y+b$ by sending their shares of $[x+a]_t, [y+b]_t$ to P_{king} and then asking P_{king} to distribute back the reconstruction results. To prevent the above mentioned double-dipping attack, $[x+a]_t$ and $[y+b]_t$ are masked by random degree-$(n-1)$ Shamir sharings of 0, denoted by $[o_1]_{n-1}$ and $[o_2]_{n-2}$. However these random masks also prevent P_{king} from detecting and error-correcting incorrect shares when reconstructing $x+a$ and $y+b$. As a result, even if a single incorrect share may cause the whole computation incorrect.

To solve this, after checking that P_{king} does not play the double-dipping attack, all parties transform $[x+a]_{n-1} := [x+a]_t + [o_1]_{n-1}$ to a tuple of 4 degree-t Shamir sharings such that each share of $[x+a]_{n-1}$ can be robustly recovered from one of the 4 newly generated degree-t Shamir sharings. This technique is known as the 4-consistent sharings [GLS19].

We note that following our approach, the above problem can be easily addressed by asking all parties send all messages when generating $[o_1]_{n-1}$ to P_{king}. In this way, P_{king} can de-mask $[x+a]_{n-1}$ and obtain $[x+a]_t$, which allows him to find incorrect shares using error correction.

2.5 Towards General Circuits

When evaluating a general circuit, one issue is to prepare the packed Shamir sharings for the next group of multiplication gates. We note that the packed Shamir sharings only allow us to do coordinate-wise multiplications. It requires that the secrets of the two input sharings are correctly aligned. This holds automatically for SIMD circuits. However, when dealing with a general circuit, the secrets may not be in the order we want. Or even worse, the secrets may not be in a single packed Shamir sharing. This problem is referred to as *network routing* [GPS21,GPS22].

In [GPS21,GPS22], the authors reduce the problem of network routing to the following sharing transformation problem. Suppose all parties hold a packed Shamir sharing $[\boldsymbol{x}]_d$ and want to perform a linear map $L : \mathbb{F}^k \to \mathbb{F}^k$ to the secrets $\boldsymbol{x}$. I.e., the goal is to compute a packed Shamir sharing $[L(\boldsymbol{x})]_d$. Following [GPS21], this task can be achieved as follows.

1. All parties prepare $([\boldsymbol{r}]_d, [L(\boldsymbol{r})]_d)$.
2. All parties compute $[\boldsymbol{x}+\boldsymbol{r}]_d$ and send their shares to P_{king}.
3. P_{king} reconstructs $\boldsymbol{x}+\boldsymbol{r}$ and distributes $[L(\boldsymbol{x}+\boldsymbol{r})]_{k-1}$.
4. All parties locally compute $[L(\boldsymbol{x})]_d = [L(\boldsymbol{x}+\boldsymbol{r})]_{k-1} - [\boldsymbol{r}]_d$.

There are two difficulties we have to address. First, how should parties prepare $([\boldsymbol{r}]_d, [L(\boldsymbol{r})]_d)$ efficiently? Indeed this is the main technical difficulty resolved in [GPS21,GPS22]. This task is not simple because each time we may need to perform a different linear map. Known solutions from [DN07,BH08] only allow us to prepare such random sharings for the same linear map many times. In [GPS22], the authors show how to perform any linear transformation efficiently (where the underlying secret sharing scheme can be any linear secret sharing scheme). Instead of using the general linear transformation, we give an efficient solution towards this task that is tailored for our case.

Second, how should parties check the correctness of P_{king}? We note that the verification protocol $\Pi_{\mathsf{VerifyPub}}$ only allows us to check the same linear maps many times, which is not sufficient since each time we may have to perform a different linear map. We design an efficient solution to resolve this issue which will be introduced later.

Efficient Preprocessing for Sharing Transformation. Our idea is to prepare random sharings $\{([\boldsymbol{r}_i]_d, [L_i(\boldsymbol{r}_i)]_d)\}_{i=1}^k$ for k different linear maps in a batch way, where recall that k is the number of secrets packed in a single sharing. Let $\boldsymbol{u}_i = L_i(\boldsymbol{r}_i)$. Then for all $j \in [k]$, $u_{i,j}$ is a linear combination of $r_{i,1}, \ldots, r_{i,k}$.

In the beginning, all parties prepare k random degree-d packed Shamir sharings $\{[\boldsymbol{r}_i]_d\}_{i=1}^k$. We may list the secrets in a matrix as follows:

$$\begin{bmatrix} r_{1,1} & \cdots & r_{1,k} \\ \vdots & \ddots & \vdots \\ r_{k,1} & \cdots & r_{k,k} \end{bmatrix}$$

Then the secrets in the same row are stored in a single packed Shamir sharing.

We observe that the packed Shamir sharings support efficient linear operations over secrets that are stored in the same positions, i.e., we can efficiently compute any linear combination of $r_{1,i}, \ldots, r_{k,i}$, which are stored at position $-i+1$. However, the sharing transformation requires us to do linear operations over secrets that are all stored in different positions. Thus, a natural idea is to re-share the matrix so that the secrets in the *same column* are stored in a single packed Shamir sharing, i.e., $\{[\boldsymbol{r}_{*,j}]_d\}_{j=1}^k$. This can be viewed as a "transpose" operation. Now all parties could efficiently compute $\{[\boldsymbol{u}_{*,j}]_d\}_{j=1}^k$. This is because the i-th secret of $[\boldsymbol{u}_{*,j}]_d$, which is $u_{i,j}$, is a linear combination of $r_{i,1}, \ldots, r_{i,k}$, which are the i-th secrets of $\{[\boldsymbol{r}_{*,j}]_d\}_{j=1}^k$. To obtain $\{[\boldsymbol{u}_i]_d\}_{i=1}^k$, we simply perform another "transpose" operation on $\{[\boldsymbol{u}_{*,j}]_d\}_{j=1}^k$.

We note that the "transpose" operation can also be viewed as one type of sharing transformation and the only difference is that it acts on the secrets of k packed Shamir sharings. Since we have to perform the same "transpose" operation many times, this can be handled efficiently by solutions from [DN07,BH08].

Efficient Verification of P_{king}. We may abstract the verification task as follows. Given $([\boldsymbol{u}]_d, [\boldsymbol{v}]_{k-1})$ and a linear map L, we want to check that $[\boldsymbol{v}]_{k-1}$ is a valid degree-$(k-1)$ packed Shamir sharing and $\boldsymbol{v} = L(\boldsymbol{u})$.

To achieve efficient verification, we follow a similar approach to that when preparing random sharings for sharing transformation. This becomes even simpler since P_{king} knows all sharings and he can help all parties perform the "transpose" operations. We sketch our solution as follows:

1. For each group of k pairs $\{([\boldsymbol{u}_i]_d, [\boldsymbol{v}_i]_{k-1}), L_i\}_{i=1}^k$, P_{king} shares the "transpose" sharings $\{[\boldsymbol{u}_{*,j}]_{k-1}\}_{j=1}^k$, $\{[\boldsymbol{v}_{*,j}]_{k-1}\}_{j=1}^k$.
2. All parties use $\Pi_{\mathsf{VerifyPub}}$ to check the correctness of all "transpose" operations. Note that these are just the same operations, which can be handled by our verification protocol.
3. All parties compute from $\{[\boldsymbol{u}_{*,j}]_{k-1}\}_{j=1}^k$ to $\{[\boldsymbol{v}_{*,j}]_{k-1}\}_{j=1}^k$ and check whether the secrets of the resulting sharings are the same as those shared by P_{king}. Again the last check can be handled by our verification protocol as well.

2.6 Summary of Our Construction

Putting all components together, we obtain a perfectly secure MPC protocol with constant online communication. In the preprocessing phase, we prepare correlated random degree-t Shamir sharings and Beaver triples. We show how to use the techniques in [BH08] to achieve this task. The communication complexity in the preprocessing phase is $O(n)$ elements per gate.

In the online phase, we follow the dispute control method and divide the circuit into $O(n^2)$ segments of equal size. Since there are at most $O(n^2)$ different dispute pairs, in the worst case, we may need to re-evaluate $O(n^2)$ segments. By having $O(n^2)$ segments, even in the worst case, the asymptotic communication complexity remains unchanged.

In each segment, we first evaluate the circuit by using packed Shamir sharings without check. This includes (1) performing proper linear transformations to prepare input packed Shamir sharings for each layer, and (2) using packed Shamir sharings to evaluate each group of gates efficiently. Note that to protect against a malicious P_{king}, every sharing that is reconstructed to P_{king} is masked by a degree-$(n-1)$ packed Shamir sharing of $\mathbf{0}$.

After evaluation, all parties together verify whether P_{king} distributes valid degree-$(k-1)$ packed Shamir sharings. After this check, all parties can reveal the mask sharing $[\mathbf{0}]_{n-1}$. Now P_{king} checks whether the shares he received are valid degree-d or degree-$(d+k-1)$ sharings, and all parties check whether P_{king} honestly follows the protocol. If all check passes, all parties proceed to the next segment. Otherwise, all parties rely on the degree-t Shamir sharings prepared in the preprocessing phase to identify a dispute pair. Then the whole segment is re-evaluated.

As a conclusion, we have the following theorem.

Theorem 1. *Let n denote the number of parties. Let $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq 2n$. For an arithmetic circuit C over $\mathbb{F}$, there exists an information-theoretic MPC protocol that computes C against a fully malicious*

adversary controlling at most $t = \frac{n-1}{3}$ *corrupted parties with perfect security. The communication cost of the protocol is expected* $O(|C| + \mathsf{Depth} \cdot n + n^5 \cdot \log n)$ *elements for the online phase and expected* $O(|C| \cdot n + \mathsf{Depth} \cdot n^2 + n^5 \cdot \log n)$ *elements for the preprocessing phase, where* Depth *is the circuit depth. The round complexity of the protocol is* $O(\mathsf{Depth} + n^2)$ *in expectation for the online phase and* $O(n^2)$ *in expectation for the preprocessing phase.*

3 Preliminary

3.1 The Model

We consider a set of n parties $\{P_1, P_2, \ldots, P_n\}$ where each party can provide inputs, receive outputs, and participate in the computation. For every pair of parties, there exists a secure (private and authenticated) 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 which can be represented as arithmetic circuits over a finite field $\mathbb{F}$ (with $|\mathbb{F}| \geq 2n$) with input, addition, multiplication, and output gates. Let $\kappa = \log |\mathbb{F}|$ be the size of an element in $\mathbb{F}$.

In this work, we consider the standard simulation-based definition of MPC [Can00]. An adversary is able to corrupt at most $t = \frac{n-1}{3}$ parties, provide inputs to corrupted parties, and receive all messages sent to the corrupted parties. Corrupted parties can deviate from the protocol arbitrarily. We denote the set of corrupted parties by $\mathcal{C}$. We consider perfect security with guaranteed output delivery. That is the protocol is guaranteed to succeed with no error.

3.2 Byzantine Agreement

Our MPC protocol uses Byzantine agreement for both broadcast and consensus. Broadcast channels are authenticated and synchronous which allow a sender to distribute a message with a guarantee that all parties will receive the same value. Consensus allows the parties who hold an individual input x_i, to reach agreement on a value x' such that $x' = x$ if every honest party holds $x_i = x$. Focusing on perfect security with corruption threshold $t < \frac{n}{3}$, to instantiate Byzantine agreement and broadcast channels, we use an expected constant round broadcast protocol proposed in [AC24] which achieves communication complexity of $O(n \cdot L)$ bits plus expected $O(n^3 \cdot \log^2 n)$ bits for broadcasting a message of size L bits.

We denote the communication complexity of a protocol by $\mathsf{P2P}(M) + N_1 \times \mathsf{BA}(L_1) + N_2 \times \mathsf{BC}(L_2)$, which means the protocol costs M bits in total over the point-to-point private channels, calls the Byzantine agreement N_1 times to reach an agreement on a message of L_1 bits, and calls the broadcast channel N_2 times with a message of L_2 bits.

3.3 Packed Shamir Secret Sharing

We 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 $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq 2n$. Let n be the number of parties and k be the number of secrets that are packed in one sharing. A degree-d ($d \geq k-1$) packed Shamir sharing with secret $\boldsymbol{x} = (x_1, \ldots, x_k) \in \mathbb{F}^k$ is a vector $(w_1, \ldots, w_n)$ for which there exists a polynomial $f(\cdot) \in \mathbb{F}[X]$ of degree at most d such that $f(-i+1) = x_i$ for all $i \in [k]$, and $f(i) = w_i$ for all $i \in \{1, 2, \ldots, n\}$. The i-th share w_i is held by party P_i. Reconstructing a degree-d packed Shamir sharing requires $d+1$ shares and can be done by Lagrange interpolation. For a random degree-d packed Shamir sharing of $\boldsymbol{x}$, any $d-k+1$ shares are independent of the secret $\boldsymbol{x}$.

In our work, we use $[\boldsymbol{x}]_d$ to denote a degree-d packed Shamir sharing of $\boldsymbol{x} \in \mathbb{F}^k$. In the following, operations (addition and multiplication) between two packed Shamir sharings are coordinate-wise. We recall two properties of the packed Shamir sharing scheme:

- Linear Homomorphism: For all $d \geq k-1$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\boldsymbol{x} + \boldsymbol{y}]_d = [\boldsymbol{x}]_d + [\boldsymbol{y}]_d$.

– Multiplicativity: Let $*$ denote the coordinate-wise multiplication operation. For all $d_1, d_2 \geq k-1$ subject to $d_1 + d_2 < n$, and for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^k$, $[\boldsymbol{x} * \boldsymbol{y}]_{d_1+d_2} = [\boldsymbol{x}]_{d_1} \cdot [\boldsymbol{y}]_{d_2}$.[1]

These two properties directly follow from computing the underlying polynomials.

Note that the second property implies that, for all $\boldsymbol{x}, \boldsymbol{c} \in \mathbb{F}^k$, all parties can locally compute $[\boldsymbol{c} * \boldsymbol{x}]_{d+k-1}$ from $[\boldsymbol{x}]_d$ and the public vector $\boldsymbol{c}$. To see this, all parties can locally transform $\boldsymbol{c}$ to degree-$(k-1)$ packed Shamir sharing $[\boldsymbol{c}]_{k-1}$. Then they can use the property of the packed Shamir sharing scheme to compute $[\boldsymbol{c} * \boldsymbol{x}]_{d+k-1} = [\boldsymbol{c}]_{k-1} \cdot [\boldsymbol{x}]_d$. This property is referred to as multiplication-friendliness in [GPS22].

Recall that t is the number of corrupted parties. Also recall that a degree-d packed Shamir secret sharing scheme is secure against $d - k + 1$ corrupted parties. In our work, we set $k = (t+2)/2 = (n+5)/6$ and $d = t + k - 1$. As we have discussed in Section 2.1, during the computation, all parties need to reconstruct degree-$(d+k-1)$ packed Shamir sharings. Using the above choices of k and d, we have $(d+k-1)+1 = t + 2k - 1 = 2t + 1$, which ensures that even if all corrupted parties have been removed when using dispute control (introduced below), we still have enough shares to reconstruct degree-$(d+k-1)$ packed Shamir sharings.

Standard Shamir Secret Sharing. When $k = 1$, $[x]_d$ is a standard degree-d Shamir sharing of $x \in \mathbb{F}$. In our work, for all $i \in \{1, \ldots, n\}$, we use $[x|i]_d$ to denote a degree-d Shamir sharing with secret stored at position $-i + 1$ rather than 0.

3.4 The Generalization of Party Elimination: Dispute Control

Our work uses the dispute control technique introduced in [BH06]. The crux for dispute control is to divide the whole computation into several segments and do it segment by segment. As for the current segment, we first evaluate it and then detect efficiently whether the evaluation is correct. It requires that at least one honest party would notice the errors in case faults occur. After detection, all parties run a consensus to agree on whether they evaluate the current segment correctly. In the case of success, all parties move to the next segment. In the case of failure, all parties run another protocol to localize a pair of two dispute parties containing at least one corrupted party. Then the current segment will be evaluated again. To avoid the dispute pairs that have been detected to disrupt the execution again, before re-evaluating the current segment, we find an intermediate party P_r which is not disputed with both P_i and P_j to help pass messages between P_i and P_j for each dispute pair (P_i, P_j) who have been already detected and are both active. Notice the secrecy is preserved because at least one of P_i and P_j is corrupted and relaying does not leak more information to the adversary. Once a party P_i is disputed with more than t parties, this party is not allowed to join future computation. Note that in this case, P_i must be a corrupted party. Then one can always find an intermediate party P_r who is not disputed with both P_i and P_j if both P_i and P_j are active. Moreover, the number of failures is bounded by $O(n^2)$ since each failure leads to finding a new dispute pair.

Hence, dividing uniformly the whole circuit into n^2 segments, each of size $|C|/n^2$, since there are totally at most n^2 failures, the whole communication complexity at most doubles.

We denote the set of currently active parties by $\mathcal{P}$ and the set of recorded dispute pairs of parties by $\mathcal{D}$. We use n' to denote the size of $\mathcal{P}$ and t' to denote the number of corrupted parties in $\mathcal{P}$. Then each time a corrupted party is eliminated, it results in $n' := n' - 1, t' := t' - 1$.

3.5 Enabling Preprocessing

Preparing Random Degree-t Shamir Sharings. Our work needs to prepare the following kinds of correlated random degree-t Shamir sharings in the preprocessing phase.

– A random degree-t Shamir sharing $[r|i]_t$. We use $\Sigma_{1,i}$ to denote this kind of random sharings.

[1] We remark that the resulting sharing $[\boldsymbol{x} * \boldsymbol{y}]_{d_1+d_2}$ is not a random degree-$(d_1 + d_2)$ secret sharing with the secret $\boldsymbol{x} * \boldsymbol{y}$.

– A pair of random degree-t Shamir sharings $([r|i]_t, [r|j]_t)$ of the same random value. We use $\Sigma_{2,i,j}$ to denote this kind of random sharings.

For all $\Sigma \in \{\Sigma_{1,i}, \Sigma_{2,i,j}\}_{i,j=1}^{n}$, we give the functionality $\mathcal{F}_{\mathsf{RandSh}}$ below that prepares N random Σ-sharings.

Functionality 1 : $\mathcal{F}_{\mathsf{RandSh}}(\Sigma)$

1. $\mathcal{F}_{\mathsf{RandSh}}$ receives a public parameter N. $\mathcal{F}_{\mathsf{RandSh}}$ also receives the set $\mathcal{C}$ of the corrupted parties' identities.
2. For all $k \in [N]$, $\mathcal{F}_{\mathsf{RandSh}}$ receives the corrupted parties' shares $\{u_j\}_{P_j \in \mathcal{C}}$ from the adversary, samples a random Σ-sharing such that for all $P_j \in \mathcal{C}$, the j-th share of the sharing is u_j, and distributes them to honest parties.

In Section D, we show that $\mathcal{F}_{\mathsf{RandSh}}$ can be realized by using techniques in [BH08] with communication complexity $\mathsf{P2P}(O(N \cdot n + n^4))$ elements plus $O(n^2) \times \mathsf{BA}(O(1))$ bits, which results in total communication of $\mathsf{P2P}(O(N \cdot n + n^5 \cdot \log n))$ elements in expectation.

Preparing Random Beaver Triples. To prepare random packed Beaver triples, we also make use of the functionality $\mathcal{F}_{\mathsf{Triples}}(i)$ that generates N random degree-t Beaver triples with secrets stored at position $-i+1$.

Functionality 2 : $\mathcal{F}_{\mathsf{Triples}}$

1. $\mathcal{F}_{\mathsf{Triples}}$ receives a public parameter N and i. $\mathcal{F}_{\mathsf{Triples}}$ also receives the set $\mathcal{C}$ of corrupted parties' identities.
2. For all $k \in [N]$:
   (a) $\mathcal{F}_{\mathsf{Triples}}$ receives from the adversary the corrupted parties' shares $\{u_j^{(1)}, u_j^{(2)}, u_j^{(3)}\}_{P_j \in \mathcal{C}}$. $\mathcal{F}_{\mathsf{Triples}}$ samples two random elements $x, y \in \mathbb{F}$ and computes $z := x \cdot y$. Then $\mathcal{F}_{\mathsf{Triples}}$ samples random degree-t Shamir secret sharings $[x|i]_t, [y|i]_t, [z|i]_t$, such that for all $P_j \in \mathcal{C}$, the j-th share of $([x|i]_t.[y|i]_t, [z|i]_t)$ is $(u_j^{(1)}, u_j^{(2)}, u_j^{(3)})$.
   (b) $\mathcal{F}_{\mathsf{Triples}}$ distributes the shares of $[x|i]_t, [y|i]_t, [z|i]_t$ to honest parties.

In Section D, we show that $\mathcal{F}_{\mathsf{Triples}}(i)$ can be realized by using techniques in [BH08] with communication complexity $\mathsf{P2P}(O(N \cdot n + n^4))$ elements plus $O(n^2) \times \mathsf{BA}(O(1))$ bits, which results in total communication of $\mathsf{P2P}(O(N \cdot n + n^5 \cdot \log n))$ elements in expectation.

4 Circuit Evaluation

Recall that, we use n to denote the number of parties, n' to denote the number of active parties and t' to denote the number of active corrupted parties. Also recall the corruption threshold $t = (n-1)/3$ and the packing parameter $k = (t+2)/2 = (n+5)/6$.

4.1 Useful Building Block for Verification

Let Σ be a linear secret sharing scheme such that a Σ sharing is fully determined by the shares of honest parties. We consider the scenario where a party P_{king} distributes N Σ-sharings, denoted by $U_1, \ldots, U_N$ to all parties (via a relay if P_{king} and P_i are disputed). These N Σ-sharings need not to be private. All parties want to check whether they hold valid Σ-sharings. We assume that an honest P_{king} always distribute valid Σ-sharings to all parties.

We introduce a functionality $\mathcal{F}_{\mathsf{VerifyPub}}$ to accomplish this task. $\mathcal{F}_{\mathsf{VerifyPub}}$ either outputs `accept` to all parties, indicating that all (honest) parties hold valid Σ-sharings, or outputs a new dispute pair that contains

at least one corrupted party. Moreover, we show that $\mathcal{F}_{\mathsf{VerifyPub}}$ can be realized with communication complexity $\mathsf{P2P}(O((N \cdot n + n^2) \cdot |\Sigma| + n^3 \cdot \log n))$ elements in expectation if the check passes in Subsection 5.1, where $|\Sigma|$ is the share size.

Functionality 3 : $\mathcal{F}_{\mathsf{VerifyPub}}(\Sigma)$

1. $\mathcal{F}_{\mathsf{VerifyPub}}$ receives the set $\mathcal{C}$ of corrupted parties' identities and the set $\mathcal{D}$ of disputed pairs.
2. $\mathcal{F}_{\mathsf{VerifyPub}}$ receives honest parties' shares of $U_1, \ldots, U_N$ from the honest parties and sends them to the adversary.
3. $\mathcal{F}_{\mathsf{VerifyPub}}$ checks if each group of shares form a valid Σ-sharing.
   - If all the checks are passed, $\mathcal{F}_{\mathsf{VerifyPub}}$ receives an instruction from the adversary:
     - If $\mathcal{F}_{\mathsf{VerifyPub}}$ receives `reject` from the adversary, it further receives a new pair of dispute parties containing at least one corrupted party and outputs `reject` together with the receiving pair to all parties.
     - If $\mathcal{F}_{\mathsf{VerifyPub}}$ receives `accept` from the adversary, it outputs `accept` to all parties.
   - If one of the checks fails, $\mathcal{F}_{\mathsf{VerifyPub}}$ receives a new pair of dispute parties containing at least one corrupted party and outputs `reject` together with the receiving pair to all parties.

4.2 Evaluating Multiplication Gates

To evaluate a group of k multiplication gates with input sharings $\{[x_i|i]_t, [y_i|i]_t\}_{i=1}^k$, we follow the technique of packed Beaver triples together with degree reduction.

In the preprocessing phase, all parties prepare random Beaver triples $\{[a_i|i]_t, [b_i|i]_t, [c_i|i]_t\}_{i=1}^k$ such that $c_i = a_i \cdot b_i$ for all $i \in [k]$, and random sharings $\{[r_i|i]_t\}_{i=1}^{2k-1}$. Before the evaluation of each segment, all parties also together prepare random degree-$(n'-1)$ packed Shamir sharings of $\mathbf{0} \in \mathbb{F}^k$. We summarize the protocol $\Pi_{\mathsf{zeroSharing}}$ below, which allows all parties to prepare $O(n)$ degree-$(n'-1)$ packed Shamir sharings of $\mathbf{0}$ with communication cost $\mathsf{P2P}(O(n^2))$ elements.

Protocol 1 : $\Pi_{\mathsf{zeroSharing}}$

1. All parties agree on a Vandermonde matrix $\mathbf{V}^{\mathrm{T}}$ of size $n' \times (n'-t')$.
2. Every party P_i randomly samples a random degree-$(n'-1)$ $\mathbf{0}$-sharing $[\mathbf{0}^{(i)}]_{n'-1}$ and distributes the shares to other parties, where $\mathbf{0}$ is of dimension k.
3. All parties locally compute

$$([\mathbf{0}_1]_{n'-1}, \ldots, [\mathbf{0}_{n'-t'}]_{n'-1})^{\mathrm{T}} = \mathbf{V} \cdot ([\mathbf{0}^{(1)}]_{n'-1}, \ldots, [\mathbf{0}^{(n')}]_{n'-1})^{\mathrm{T}}.$$

We give the description of Π_{Mult} below. The communication complexity per batch of k multiplication gates is $\mathsf{P2P}(O(n))$ elements.

Protocol 2 : Π_{Mult}

For a batch of k multiplication gates, all parties hold input sharings $\{[x_i|i]_t, [y_i|i]_t\}_{i=1}^k$. In addition, all parties hold the following sharings prepared in the preprocessing phase.

- A group of Beaver triples $\{[a_i|i]_t, [b_i|i]_t, [c_i|i]_t\}_{i=1}^k$;
- A group of random degree-t Shamir sharings $\{[r_i|i]_t\}_{i=1}^{2k-1}$;

All parties also hold the following sharings prepared right before evaluating the current segment: three degree-$(n'-1)$ packed Shamir sharings $[\mathbf{0}_1]_{n'-1}, [\mathbf{0}_2]_{n'-1}, [\mathbf{0}_3]_{n'-1}$. All parties run the following steps to compute $\{[z_i|i]_t\}_{i=1}^k$.

1. All parties locally compute $[\boldsymbol{x}]_{t+k-1}, [\boldsymbol{y}]_{t+k-1}, [\boldsymbol{a}]_{t+k-1}, [\boldsymbol{b}]_{t+k-1}, [\boldsymbol{c}]_{t+k-1}$ from $\{[x_i|i]_t, [y_i|i]_t\}_{i=1}^k$ and $\{[a_i|i]_t, [b_i|i]_t, [c_i|i]_t\}_{i=1}^k$.

2. All parties locally compute $[\boldsymbol{x}+\boldsymbol{a}]_{n'-1} = [\boldsymbol{x}]_{t+k-1} + [\boldsymbol{a}]_{t+k-1} + [\mathbf{0}_1]_{n'-1}$ and $[\boldsymbol{y}+\boldsymbol{b}]_{n'-1} = [\boldsymbol{y}]_{t+k-1} + [\boldsymbol{b}]_{t+k-1} + [\mathbf{0}_2]_{n'-1}$, and send their shares to P_{king}.
3. P_{king} reconstructs $\boldsymbol{x}+\boldsymbol{a}$ and $\boldsymbol{y}+\boldsymbol{b}$, distributes $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{k-1}$ to all parties.
4. All parties locally compute

$$[\boldsymbol{z}]_{t+2k-2} = [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{b}]_{t+k-1}$$

$$-[\boldsymbol{a}]_{t+k-1} \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} + [\boldsymbol{c}]_{t+k-1}$$

   and a random degree-$(t+2k-2)$ packed Shamir sharing $[\boldsymbol{r}]_{t+2k-2}$ from $\{[r_i|i]_t\}_{i=1}^{2k-1}$.
   Finally, all parties locally compute $[\boldsymbol{z}+\boldsymbol{r}]_{n'-1} = [\boldsymbol{z}]_{t+2k-2} + [\boldsymbol{r}]_{t+2k-2} + [\mathbf{0}_3]_{n'-1}$ and send their shares to P_{king}.
5. P_{king} reconstructs $\boldsymbol{z}+\boldsymbol{r}$ and distributes $[\boldsymbol{z}+\boldsymbol{r}]_{k-1}$ to all parties.
6. All parties locally compute $[z_i|i]_t = [\boldsymbol{z}+\boldsymbol{r}]_{k-1} - [r_i|i]_t$ for all $i \in [k]$.

4.3 Handling Sharing Transformations

In this section, we show how to efficiently perform sharing transformations. This will be an important component to evaluate a general circuit using packed Shamir sharings. The task of sharing transformation is to transform a degree-$(t+k-1)$ packed Shamir sharing $[\boldsymbol{x}]_{t+k-1}$ to $[L(\boldsymbol{x})]_{t+k-1}$, where $L : \mathbb{F}^k \to \mathbb{F}^k$ is a linear map.

Preparing Correlated Random Sharings for Linear Maps. As we discussed in Subsection 2.5, the sharing transformation is done by first preparing a pair of random sharings $([\boldsymbol{r}]_{t+k-1}, [L(\boldsymbol{r})]_{t+k-1})$.

We first introduce a protocol $\Pi_{\mathsf{Transpose}}$ that allows all parties to transform $\{[x_{i,j}|j]_t\}_{i,j=1}^k$ to $\{[x_{j,i}|j]\}_{i,j=1}^k$. The communication complexity of $\Pi_{\mathsf{Transpose}}$ is $\mathsf{P2P}(O(n^2))$ elements before running the consensus in step 6 in the case of success. Note that all parties run the consensus on their `happy-bit`s only once after they finish all the executions of the protocol $\Pi_{\mathsf{Transpose}}$ to perform the 'transpose' operations in each segment.

Protocol 3 : $\Pi_{\mathsf{Transpose}}$ in $\mathcal{F}_{\mathsf{VerifyPub}}$-hybrid model

All parties hold input sharings $\{[x_{i,j}|j]_t\}_{i,j=1}^k$. All parties in addition hold the following random sharings prepared in the preprocessing phase: $\{[r_{i,j}|j]_t, [r_{j,i}|j]_t\}_{i,j=1}^k$. All parties run the following steps to compute $\{[x_{j,i}|j]_t\}_{i,j=1}^k$.

1. For all $i \in [k]$, all parties locally compute $[\boldsymbol{x}_i]_{t+k-1}$ and $[\boldsymbol{r}_i]_{t+k-1}$ from $\{[x_{i,j}|j]_t\}_{j=1}^k$ and $\{[r_{i,j}|j]_t\}_{j=1}^k$.
2. For all $i \in [k]$, all parties locally compute $[\boldsymbol{x}_i + \boldsymbol{r}_i]_{t+k-1} = [\boldsymbol{x}_i]_{t+k-1} + [\boldsymbol{r}_i]_{t+k-1}$ and send their shares to P_{king}.
3. P_{king} checks whether all received degree-$(t+k-1)$ packed Shamir sharings are valid. If not, supposing the i-th sharing is inconsistent, P_{king} broadcasts a complain to all parties. Then all parties send their shares of $\{[x_{i,j}+r_{i,j}|j]_t\}_{j=1}^k$ to P_{king}.
   P_{king} reconstructs $(x_{i,j}+r_{i,j})_{j=1}^k$ and computes $[(x_{i,j}+r_{i,j})_{j=1}^k]_{t+k-1}$. Then P_{king} identifies the cheating party P_i and broadcasts (i, x, x'), where P_i should have sent x to P_{king}, but P_{king} receives $x' \neq x$. Then P_i and the relay between P_i and P_{king} broadcast the messages they believe. Set two adjacent parties broadcasting differently to be the dispute pair.
4. Otherwise, P_{king} reconstructs $(x_{i,j}+r_{i,j})_{i,j=1}^k$. For all $i \in [k]$, P_{king} distributes $[\boldsymbol{x}_{*,i} + \boldsymbol{r}_{*,i}]_{k-1}$ to all parties.
5. For all $i, j \in [k]$, all parties locally compute $[x_{j,i}|j]_t = [\boldsymbol{x}_{*,i} + \boldsymbol{r}_{*,i}]_{k-1} - [r_{j,i}|j]_t$.
6. All parties invoke $\mathcal{F}_{\mathsf{VerifyPub}}$ to check whether P_{king} correctly distributes valid degree-$(k-1)$ packed Shamir sharings. If not, all parties take a new dispute pair as output. Otherwise, all parties take $\{[x_{j,i}|j]_t\}_{i,j=1}^k$ as output.

We will use $\Pi_{\mathsf{Transpose}}$ to prepare correlated random sharings for sharing transformations in $\Pi_{\mathsf{PrepTrans}}$. The communication complexity of $\Pi_{\mathsf{PrepTrans}}$ is $\mathsf{P2P}(O(N \cdot n + n^3))$ elements plus $O(1) \times \mathsf{BA}(O(1))$ bits accounting for total communication of expected $\mathsf{P2P}(O(N \cdot n + n^3 \cdot \log n))$ elements to prepare random sharings for N linear transformations if succeeds.

Protocol 4 : $\Pi_{\mathsf{PrepTrans}}$ **in** $\mathcal{F}_{\mathsf{VerifyPub}}$**-hybrid model**

All parties together hold N linear maps $L_1, \ldots, L_N$ and the goal is to prepare $\{[r_{i,j}|j]_t, [L_{i,j}(\boldsymbol{r}_i)|j]_t\}_{j=1}^k$ for all $i \in [N]$. Here $L_{i,j}(\cdot)$ denote the linear function that outputs the j-th value of $L_i(\cdot)$.
For each group of k linear maps, say $L_1, \ldots, L_k$, in the beginning, all parties hold the following random sharings prepared in the preprocessing phase: $\{[r_{i,j}|j]_t, [r_{j,i}|j]_t\}_{i,j=1}^k$. All parties do the following.

1. All parties locally compute
$$[L_{j,i}(\boldsymbol{r}_j)|j]_t = L_{j,i}([r_{j,1}|j]_t, \ldots, [r_{j,k}|j]_t), \forall i, j \in [k].$$
2. All parties invoke $\Pi_{\mathsf{Transpose}}$ on $\{[L_{j,i}(\boldsymbol{r}_j)|j]_t\}_{i,j=1}^k$ to obtain shares of $\{[L_{i,j}(\boldsymbol{r}_i)|j]_t\}_{i,j=1}^k$.

Finally, if all parties succeed in $\Pi_{\mathsf{Transpose}}$, all parties invoke $\mathcal{F}_{\mathsf{VerifyPub}}$ to check whether P_{king} correctly performs the transpose operation in $\Pi_{\mathsf{Transpose}}$. This is done as follows. In $\Pi_{\mathsf{Transpose}}$, P_{king} receives $\{[\boldsymbol{x}_i + \boldsymbol{r}_i]_{t+k-1}\}_{i=1}^k$ from all parties and distributes $\{[\boldsymbol{x}_{*,i} + \boldsymbol{r}_{*,i}]_{k-1}\}_{i=1}^k$ to all parties. We may effectively think that P_{king} distributes $\{[\boldsymbol{x}_i + \boldsymbol{r}_i]_{t+k-1}\}_{i=1}^k$ and $\{[\boldsymbol{x}_{*,i} + \boldsymbol{r}_{*,i}]_{k-1}\}_{i=1}^k$ to all parties. All parties invoke $\mathcal{F}_{\mathsf{VerifyPub}}$ to check the correctness of the transpose operations.

- If the check fails, all parties take a new dispute pair as output. Otherwise, all parties take $\{[r_{i,j}|j]_t, [L_{i,j}(\mathbf{r}_i)|j]_t\}_{i,j=1}^{N,k}$ as output.

Performing Sharing Transformations. With $\{[r_j|j]_t, [L_j(\boldsymbol{r})|j]_t\}_{j=1}^k$, we show how to transform $\{[x_j|j]_t\}_{j=1}^k$ to $\{[L_j(\boldsymbol{x})|j]_t\}_{j=1}^k$ in Π_{ShTrans}. The communication complexity of Π_{ShTrans} is $\mathsf{P2P}(O(n))$ elements.

Protocol 5 : Π_{ShTrans}

All parties take $\{[x_j|j]_t\}_{i=1}^k$ and a linear map $L = (L_1, \ldots, L_k)$ as input. All parties also hold the following random sharings prepared right before the evaluation of the current segment:

- Two groups of correlated random sharings $\{[r_j|j]_t, [L_j(\boldsymbol{r})|j]_t\}_{j=1}^k$,
- A degree-$(n'-1)$ packed Shamir sharing $[\mathbf{0}]_{n'-1}$.

All parties run the following steps to compute $\{[L_j(\boldsymbol{x})|j]_t\}_{j=1}^k$.

1. All parties locally compute $[\boldsymbol{x}]_{t+k-1}$ and $[\boldsymbol{r}]_{t+k-1}$ from $\{[x_j|j]_t\}_{j=1}^k$ and $\{[r_j|j]_t\}_{j=1}^k$.
2. All parties locally compute $[\boldsymbol{x}+\boldsymbol{r}]_{n'-1} = [\boldsymbol{x}]_{t+k-1} + [\boldsymbol{r}]_{t+k-1} + [\mathbf{0}]_{n'-1}$ and send their shares to P_{king}.
3. P_{king} reconstructs $\boldsymbol{x}+\boldsymbol{r}$, computes $L(\boldsymbol{x}+\boldsymbol{r})$, and distributes $[L(\boldsymbol{x}+\boldsymbol{r})]_{k-1}$ to all parties.
4. All parties locally compute $[L_j(\boldsymbol{x})|j]_t = [L(\boldsymbol{x}+\boldsymbol{r})]_{k-1} - [L_j(\boldsymbol{r})|j]_t$ for all $j \in [k]$.

4.4 Summary of the Evaluation Phase

To compute a general circuit with packed Shamir sharings, we utilize the techniques in [GPS21] for network routing. At a high level, for any circuit C, we first divide gates of the same type in each layer into groups of size k. After we obtain output packed Shamir sharings for each group of gates in the current layer, the authors in [GPS21] show that the input packed Shamir sharings in the next layer can be obtained as follows:

– For each output packed Shamir sharings in the current layer, we perform the fan-out operation to copy each secret enough number of times. For example, if a packed Shamir sharing $[x_1, x_2, x_3]_d$ satisfies that x_1, x_2, x_3 will be used by $(2, 3, 1)$ times in future layers, then all parties compute $[x_1, x_1, x_2]_d$ and $[x_2, x_2, x_3]_d$. Note that each new packed Shamir sharing can be obtained by performing a proper linear transformation to the original packed Shamir sharing.
– After the fan-out operations, for each obtained packed Shamir sharing, perform a proper permutation on the secrets, which is also a linear transformation.
– After doing the above two steps, we move to prepare the input packed Sharings we need in the next layer. The main property that is achieved in [GPS21] is that, now for every packed Shamir sharing $[\boldsymbol{x}]_{t+k-1}$ we want to prepare, the previous steps have generated k packed Shamir sharings $\{[\boldsymbol{x}^{(i)}]_{t+k-1}\}_{i=1}^{k}$ such that there exists a permutation $p : \{1, \ldots, k\} \to \{1, \ldots, k\}$ and $x_i = x^{(i)}_{p(i)}$. In other words, all secrets we want to collect all come from different positions. In this way, all parties can efficiently collect secrets they want and obtain $[\boldsymbol{x}']_{t+k-1}$ without changing the positions of the secrets, i.e., $x_i = x^{(i)}_{p(i)} = x'_{p(i)}$.
We note that in our case this task is even simpler. This is because for every $[\boldsymbol{x}^{(i)}]_{t+k-1}$, we actually prepare $\{[x^{(i)}_j|j]_t\}_{j=1}^{k}$. Thus, we just need to pick $\{[x_i|p(i)]_t\}_{i=1}^{k} = \{x^{(i)}_{p(i)}|p(i)\}_{i=1}^{k}$.
– Finally, to obtain $[\boldsymbol{x}]_{t+k-1}$, we permute the secrets in $[\boldsymbol{x}']_{t+k-1}$, which again is a linear transformation.

Since our work uses [GPS21] in a black box way, we refer the readers to [GPS21] for how to find these linear maps.

We assume that for output packed Shamir sharings from the last segment, the first two steps have been done. We will maintain the invariant in the current segment. We summarize our evaluation protocol as Π_{Eval} below and analyze the number of required different kinds of preprocessing data as follows, where we suppose there are N linear maps and M multiplication gates needed to be handled in this segment and thus $N = O(|C|/n^3), M = O(|C|/n^2)$.

– A degree-$(n'-1)$ packed Shamir sharing $[\mathbf{0}]_{n'-1}$: $\frac{3M}{k} + N$.
– A group of Beaver triples $\{[a_i|i]_t, [b_i|i]_t, [c_i|i]_t\}_{i=1}^{k}$: $\frac{M}{k}$ groups.
– A group of random degree-t Shamir sharings $\{[r_i|i]_t\}_{i=1}^{2k-1}$: $\frac{M}{k}$ groups.
– A group of Shamir sharings $\{[r_{i,j}|j]_t, [r_{j,i}|j]_t\}_{i,j=1}^{k}$: $\frac{2N}{k}$ groups.

Protocol 6 : Π_{Eval}

For each group of k wires before the current segment, all parties hold $\{[x_i|i]_t\}_{i=1}^{k}$. All parties run the following steps.

1. Suppose the linear maps we need to perform in this segment is $L_1, \ldots, L_N$. All parties invoke $\Pi_{\mathsf{PrepTrans}}$ to prepare $\{[r_{i,j}|j]_t, [L_{i,j}(\boldsymbol{r}_i)|j]_t\}_{i\in[N],j\in[k]}$. All parties invoke $\Pi_{\mathsf{zeroSharing}}$ to prepare $[\mathbf{0}]_{n'-1}$.
2. All parties evaluate the current segment layer by layer.
   (a) For each group of input wires $\boldsymbol{x}$, all parties locally collect secrets and obtain $\{[x_i|p(i)]_t\}_{i=1}^{k}$, where p is a permutation over $[k]$. This step is guaranteed by the network routing protocol in [GPS21].
   (b) For each group of input wires $\boldsymbol{x}$, All parties invoke Π_{ShTrans} on $\{[x_i|p(i)]_t\}_{i=1}^{k}$ to obtain $\{[x_i|i]_t\}_{i=1}^{k}$.
   (c) For each group of addition gates with input packed Shamir sharings $\{[x_i|i]_t\}_{i=1}^{k}$ and $\{[y_i|i]_t\}_{i=1}^{k}$, all parties locally compute $[z_i|i]_t = [x_i|i]_t + [y_i|i]_t$ via local computation for $i \in [k]$.
   (d) For each group of multiplication gates with input packed Shamir sharings $\{[x_i|i]_t\}_{i=1}^{k}$ and $\{[y_i|i]_t\}_{i=1}^{k}$, all parties invoke Π_{Mult} to compute $\{[z_i|i]_t\}_{i=1}^{k}$.
   (e) For each output packed Shamir sharing $\{[z_i|i]_t\}_{i=1}^{k}$ and the linear map L that is needed to perform on $\boldsymbol{z}$, all parties invoke Π_{ShTrans} on $\{[z_i|i]_t\}_{i=1}^{k}$ to obtain $\{[L_i(\boldsymbol{z})|i]_t\}_{i=1}^{k}$. (Fan-out gates and permutations are handled here.)

5 Efficient Verification

In this section, we study how to perform efficient verification after completing evaluating the current segment and handling sharing transformations.

5.1 Instantiating Batch-wise Verification

To achieve perfect security with constant online communication complexity, we need an efficient protocol to realize $\mathcal{F}_{\mathsf{VerifyPub}}$ introduced in Subsection 4.1 which allows to verify linear secret sharings distributed by party P_{king} who is responsible for the consistency of these sharings. Recall the verification either is passed only if the shares held by all honest parties are consistent, or fails and further provides a new dispute pair that contains at least one corrupted party.

We follow the high-level idea in Section 2.2 and give the construction below. As for its communication cost, notice in the case that faults occur, the party P_r will broadcast an inconsistent message he finds which is in the form of $(\ell, P_i, P_j, v, v', \text{disputed})$ with ℓ denoting the index of the message in step 4b of the protocol $\Pi_{\mathsf{VerifyPub}}$. Using an expected constant round batch broadcast protocol like [AC24] leads to expected communication of $O(n \cdot \log M + n^3 \cdot \log^2 n)$ bits, where M denotes the number of all messages in this segment. Since $n \cdot \log M \leq M + n^3 \cdot \log^2 n$, this part of communication will not dominate the whole communication complexity.

The communication complexity of $\Pi_{\mathsf{VerifyPub}}$ is $\mathsf{P2P}(O((N \cdot n + n^2) \cdot |\Sigma|))$ elements plus $\mathsf{BA}(O(1))$ bits resulting in total communication of $\mathsf{P2P}(O((N \cdot n + n^2) \cdot |\Sigma| + n^3 \cdot \log n))$ elements in expectation to check N Σ-sharings if the check passes.

Protocol 7 : $\Pi_{\mathsf{VerifyPub}}(\Sigma)$

All parties hold N Σ-sharings $U_1, \ldots, U_N$, which are distributed by P_{king}. For each group of $2t+1$ sharings, say $U_1, \ldots, U_{2t+1}$. Let $T = 2t + 1$. All parties do the following.

1. All parties agree on n' distinct field elements $\{\beta_i\}_{i=1}^{n'}$. Define $U(x) = \sum_{h=1}^{T} U_h x^{h-1}$. For all $i \in [n']$, every party P_j computes its share of $U(\beta_i)$ and sends it to P_i.
2. For all $i \in [n']$, party P_i checks if the shares it receives form a Σ-sharing. If they are inconsistent, P_i gets `unhappy`.
3. Fault Detection: Reach an agreement on whether at least one party is `unhappy`.
   (a) Every party P_i sends its happy-bit together with its index (P_i, `happy-bit`) to every party P_j, who gets `unhappy` if at least one P_i claims to be `unhappy`.
   (b) All the parties run a consensus protocol on their respective happy-bits. If the consensus outputs `happy`, all parties output `accept` and halt. Otherwise, the following Fault-Localization step is executed.
4. Fault Localization: Localize a pair of parties E in which at least one party is corrupted.
   If P_r sends (P_r, `unhappy`), P_r checks the consistency as follows:
   (a) All parties send to P_r their shares received from P_{king}. P_{king} sends to P_r the shares it distributed. (Note if any party conflicts with P_r, then this will be relayed by an intermediate party.) All parties also send to P_r all messages they sent and received in the Fault Detection step.
   (b) If P_i and P_j do not agree on some message, P_r broadcasts $(\ell, P_i, P_j, v, v', \text{disputed})$, where ℓ is the index of a message where P_i should have sent v to P_j, while P_j claimed to have received $v' \neq v$. Go to step 4(e).
   (c) If P_{king} itself sends inconsistent messages, P_r broadcasts $(P_{\text{king}}, \text{corrupt})$. Go to step 4(e).
   (d) If some P_j sends an incorrect share of $U(\beta_r)$ or sends an incorrect `happy` bit, P_r broadcasts $(P_j, \text{corrupt})$. Go to step 4(e).
   (e) All parties localize a dispute pair:
      i. If a pair of parties that has already been broadcast, then P_r is corrupted and will be eliminated.

ii. If $(P_j, \text{corrupt})$ is broadcast, then the relay for (P_j, P_r) together with P_j broadcast whether they agree with P_r. Denote the relay by $P_{j\leftrightarrow r}$. If P_j agrees, then P_j is corrupted. If $P_j, P_{j\leftrightarrow r}$ have different opinions, set $E = \{P_j, P_{j\leftrightarrow r}\}$. Otherwise, set $E = \{P_r, P_{j\leftrightarrow r}\}$

iii. Otherwise when $(\ell, P_i, P_j, v, v', \text{disputed})$ is broadcast, all the parties relaying for $(P_r, P_i), (P_i, P_j), (P_j, P_r)$ together with P_i and P_j broadcast what the message they believe is correct. Set E to be the set of the two adjacent parties whose opinions are different.

Lemma 1. *After finishing step 1 and step 2 in $\Pi_{\mathsf{VerifyPub}}$, at least one honest party will get* **unhappy** *if the shares of U_h belonging to honest parties are inconsistent for some $1 \le h \le T$.*

See the proof in Appendix A.

Lemma 2. *The protocol $\Pi_{\mathsf{VerifyPub}}$ securely computes $\mathcal{F}_{\mathsf{VerifyPub}}$ against a fully malicious adversary who controls $t' \le (n'-1)/3$ parties.*

See the proof in Appendix B.

5.2 Verifying Multiplication Gates

For now, all parties already complete the computation of the current segment including evaluating all groups of multiplication gates inside the current segment and handling sharing transformations.

To verify the evaluation above, all parties first invoke $\mathcal{F}_{\mathsf{VerifyPub}}$ to check whether P_{king} distributes valid degree-$(k-1)$ packed Shamir sharings in Π_{Mult}. If the verification passes, all parties open their shares of degree-$(n'-1)$ packed Shamir sharings of $\mathbf{0}$ prepared in $\Pi_{\mathsf{zeroSharing}}$. After subtracting the shares of degree-$(n'-1)$ packed Shamir sharings of $\mathbf{0}$, P_{king} obtains the original messages receiving from other parties

1. two degree-$(t+k-1)$ sharings $[\boldsymbol{x}+\boldsymbol{a}]_{t+k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{t+k-1}$ and
2. a degree-$(t+2k-2)$ sharing $[\boldsymbol{z}+\boldsymbol{r}]_{t+2k-2}$.

Notice P_{king} is able to detect whether other parties send these shares correctly by simply checking if the received shares lie on a degree-$(t+k-1)$ polynomial for the first case and a degree-$(t+2k-2)$ polynomial for the second case because for both cases, shares of honest parties fully determine the whole sharing. In the case of inconsistency, P_{king} is supposed to localize a dispute pair. For the first case, all parties send their degree-t shares of $\{[x_j|j]_t + [a_j|j]_t\}_{j=1}^k$ to P_{king} who takes advantage of error correction code to robustly reconstruct $\boldsymbol{x}+\boldsymbol{a}$, and compares them with $[\boldsymbol{x}+\boldsymbol{a}]_{t+k-1}$ to detect who is cheating (note at least $2t+1$ shares of $[\boldsymbol{x}+\boldsymbol{a}]_{t+k-1}$ held by honest parties correctly share the secret $\boldsymbol{x}+\boldsymbol{a}$).

As for the second case, we follow the high-level idea in Section 2.3 to find a dispute pair.

The protocol $\Pi_{\mathsf{VerifyMult}}$ described below requires all parties to prepare randomness $\{[a'_j|j]_t, [b'_j|j]_t, [c'_j|j]_t\}_{j=1}^k$, $\{[r'_j|j]_t\}_{j=1}^{2k-1}$ in the preprocessing phase. The communication of verifying N multiplication gates is $\mathsf{P2P}(O(N+n))$ elements plus $O(1)\times\mathsf{BA}(O(1))$ bits accounting for total communication of expected $\mathsf{P2P}(O(N+n^3\cdot\log n))$ elements in the case of success.

Protocol 8 : $\Pi_{\mathsf{VerifyMult}}$ in $\mathcal{F}_{\mathsf{VerifyPub}}$-hybrid model

All parties hold random degree-t Shamir sharings prepared in the preprocessing phase $\{[a'_j|j]_t, [b'_j|j]_t, [c'_j|j]_t\}_{j=1}^k$ and $\{[r'_j|j]_t\}_{j=1}^{2k-1}$.

1. All parties invoke $\mathcal{F}_{\mathsf{VerifyPub}}$ to check whether P_{king} sends valid degree-$(k-1)$ packed Shamir sharings in step 3 and step 5 of Π_{Mult}.
   - If the verification passes, then all parties send all messages in $\Pi_{\mathsf{zeroSharing}}$ to P_{king}. Then P_{king} checks whether all parties honestly follow $\Pi_{\mathsf{zeroSharing}}$.

- If not, P_{king} either broadcasts (ℓ, v, v', P_i, P_j) if P_i and P_j do not agree on the same message, or $(P_i, \text{corrupt})$ if P_i does not follow the protocol. Then follow Step 4.(e) in $\Pi_{\mathsf{VerifyPub}}$ to identify a new dispute pair. All parties take the new dispute pair as output.
- Otherwise P_{king} subtracts the degree-$(n'-1)$ packed Shamir sharings of $\mathbf{0}$ from the sharings he received in Π_{Mult}.

– Otherwise, all parties take the new dispute pair as output.

2. P_{king} checks if one of other parties sends wrong share $[\boldsymbol{x}+\boldsymbol{a}]_{t+k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{t+k-1}$ in step 2 of Π_{Mult}: P_{king} only needs to check whether these shares are of degree $t+k-1$.
   – If it is inconsistent, every party P_i sends $\{[x_j|j]_t + [a_j|j]_t, [y_j|j]_t + [b_j|j]_t\}_{j=1}^k$ to P_{king}, $\forall i \in [n']$. P_{king} reconstructs $\boldsymbol{x}+\boldsymbol{a}, \boldsymbol{y}+\boldsymbol{b}$, compares it with $[\boldsymbol{x}+\boldsymbol{a}]_{t+k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{t+k-1}$ to detect who is cheating and broadcasts (i, x, x'), where P_i should have sent x to P_{king}, but P_{king} claims to have received $x' \neq x$. P_{king} asks P_i and their relay to broadcast the messages they believe. Set two adjacent parties broadcasting differently to be the dispute pair. All parties take the new dispute pair as output.
3. All parties invoke $\mathcal{F}_{\mathsf{VerifyPub}}$ to check whether P_{king} sends degree-$(k-1)$ sharings with correct secret value $\boldsymbol{x}+\boldsymbol{a}, \boldsymbol{y}+\boldsymbol{b}$ in step 3 of Π_{Mult}.
   – If not, all parties take the new dispute pair as output.
4. P_{king} checks if it receives a valid deg-$(t+2k-2)$ Shamir secret sharing in step 4 of Π_{Mult}.
   – If it is inconsistent, all parties localize a dispute pair as follows, using fresh random sharings $\{[a'_j|j]_t, [b'_j|j]_t, [c'_j|j]_t\}_{j=1}^k$ and $\{[r'_j|j]_t\}_{j=1}^{2k-1}$.
   (a) Every party locally computes a degree-$(t+2k-2)$ sharing $[\boldsymbol{v}]_{t+2k-2}$ as

$$\begin{aligned}[\boldsymbol{v}]_{t+2k-2} = [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} - [\boldsymbol{x}+\boldsymbol{a}]_{k-1} \cdot [\boldsymbol{b'}]_{t+k-1} \\ -[\boldsymbol{a'}]_{t+k-1} \cdot [\boldsymbol{y}+\boldsymbol{b}]_{k-1} + [\boldsymbol{c'}]_{t+k-1} + [\boldsymbol{r'}]_{t+2k-2},\end{aligned}$$

   where $[\boldsymbol{a'}]_{t+k-1}, [\boldsymbol{b'}]_{t+k-1}, [\boldsymbol{c'}]_{t+k-1}, [\boldsymbol{r'}]_{t+2k-2}$ are locally computed from $\{[a'_j|j]_t, [b'_j|j]_t, [c'_j|j]_t\}_{j=1}^k$ and $\{[r'_j|j]_t\}_{j=1}^{2k-1}$.
   Then all parties send $[\boldsymbol{v}]_{t+2k-2}$ to P_{king}.
   (b) P_{king} checks if the received shares in step 4(a) satisfy a degree-$(t+2k-2)$ Shamir secret sharing.
   - If it is inconsistent, every party P_i sends $\{[a'_j|j]_t, [b'_j|j]_t, [c'_j|j]_t\}_{j=1}^k$ and $\{[r'_j|j]_t\}_{j=1}^{2k-1}$ to P_{king} who localizes a dispute pair as follows. P_{king} reconstructs all degree-t Shamir sharings and computes $[\boldsymbol{a'}]_{t+k-1}, [\boldsymbol{b'}]_{t+k-1}, [\boldsymbol{c'}]_{t+k-1}, [\boldsymbol{r'}]_{t+2k-2}$. Then P_{king} computes $[\boldsymbol{v}]_{t+2k-2}$ and detect who is cheating. The rest is the same as step 2.
   (c) If it is consistent, every party P_i sends $\{[a'_j|j]_t + [a_j|j]_t, [b'_j|j]_t + [b_j|j]_t, [c'_j|j]_t + [c_j|j]_t\}_{j=1}^k$ and $\{[r'_j|j]_t + [r_j|j]_t\}_{j=1}^{2k-1}$ to P_{king} who localizes a dispute pair as in step 4(b).
5. All parties invoke $\mathcal{F}_{\mathsf{VerifyPub}}$ to check whether P_{king} sends a degree-$(k-1)$ sharing with correct secret value $\boldsymbol{z}+\boldsymbol{r}$ in step 5 of Π_{Mult}.
   – If not, all parties take the new dispute pair as output.

5.3 Verifying Sharing Transformations

Following the verification of multiplication gates, after degree-$(n'-1)$ packed Shamir sharings of $\mathbf{0}$ are opened, P_{king} checks whether all parties send correct shares $[\boldsymbol{x}+\boldsymbol{r}]_{t+k-1}$. Then all parties check whether P_{king} honestly shares $[L(\boldsymbol{x}+\boldsymbol{r})]_{k-1}$.

The protocol $\Pi_{\mathsf{VerifyShTrans}}$ is described below. The communication of verifying N sharing transformations is $\mathsf{P2P}(O(N \cdot n + n^2))$ elements plus $O(1) \times \mathsf{BA}(O(1))$ bits accounting for total communication of expected $\mathsf{P2P}(O(N \cdot n + n^3 \cdot \log n))$ elements in the case of success.

Protocol 9 : $\Pi_{\mathsf{VerifyShTrans}}$ in $\mathcal{F}_{\mathsf{VerifyPub}}$-hybrid model

1. All parties check whether P_{king} distributes a degree-$(k-1)$ Shamir sharing in step 3 of Π_{ShTrans} in the same way as step 1 in $\Pi_{\mathsf{VerifyMult}}$. Then P_{king} checks if one of other parties sends wrong share of $[\boldsymbol{x}+\boldsymbol{r}]_{t+k-1}$ in step 2 of Π_{ShTrans} in the same way as step 2 in $\Pi_{\mathsf{VerifyMult}}$.
2. All parties check whether P_{king} sends valid sharing of degree $k-1$ with correct secret $L(\boldsymbol{x}+\boldsymbol{r})$ as follows. For each group of k linear transformations, say $L_1, L_2, \ldots, L_k$, all parties hold $\{[\boldsymbol{x}_i+\boldsymbol{r}_i]_{t+k-1}, [L_i(\boldsymbol{x}_i+\boldsymbol{r}_i)]_{k-1}\}_{i=1}^k$. Let $[\boldsymbol{u}_i]_{t+k-1}$ denote $[\boldsymbol{x}_i+\boldsymbol{r}_i]_{t+k-1}$ and $[\boldsymbol{v}_i]_{k-1}$ denote $[L_i(\boldsymbol{x}_i+\boldsymbol{r}_i)]_{k-1}$.
   (a) P_{king} distributes $\{[\boldsymbol{u}_{*,i}]_{k-1}\}_{i=1}^k$ and $\{[\boldsymbol{v}_{*,i}]_{k-1}\}_{i=1}^k$ to all parties.
   (b) All parties invoke $\mathcal{F}_{\mathsf{VerifyPub}}$ to check whether P_{king} honestly perform the transpose operations for $\{[\boldsymbol{u}_i]_{t+k-1}\}_{i=1}^k$. This step is done for all groups of k linear transformations.
      – If not, all parties take a new dispute pair as output.
   (c) All parties invoke $\mathcal{F}_{\mathsf{VerifyPub}}$ to check whether P_{king} honestly perform the transpose operations for $\{[\boldsymbol{v}_i]_{k-1}\}_{i=1}^k$. This step is done for all groups of k linear transformations.
      – If not, all parties take a new dispute pair as output.
   (d) For all $i \in [k]$, all parties locally compute

$$[\boldsymbol{o}_i]_{2k-2} = [\boldsymbol{v}_{*,i}]_{k-1} - \sum_{j=1}^{k} [\boldsymbol{e}_j]_{k-1} \cdot L_{j,i}([\boldsymbol{u}_{*,1}]_{k-1}, \ldots, [\boldsymbol{u}_{*,k}]_{k-1}),$$

   where $\boldsymbol{e}_j \in \mathbb{F}^k$ is the j-th unit vector (i.e., the j-th value is 1 while all other values are 0). Note that P_{king} can also compute $\{[\boldsymbol{o}_i]_{2k-2}\}_{i=1}^k$. Effectively, we view these sharings are distributed by P_{king}.
   (e) All parties invoke $\mathcal{F}_{\mathsf{VerifyPub}}$ to check whether $\{[\boldsymbol{o}_i]_{2k-2}\}_{i=1}^k$ are degree-$(2k-2)$ packed secret sharings of $\mathbf{0}$.
      – If not, all parties take a new dispute pair as output.

6 Main Protocol with Perfect Security

In this section, we will conclude our main protocol and discuss the instantiation of preprocessing phase.

6.1 Input Layer

We sketch the protocol Π_{Input} that allows a party P_s to share his inputs to all parties. At the end of Π_{Input}, either all parties hold degree-t Shamir sharings of P_s's input, or a new dispute pair is identified and P_s will re-share his input.

For each group of k inputs $\boldsymbol{x}$ of P_s, all parties prepare random degree-t Shamir sharings $\{[r_j|j]_t\}_{j=1}^k$ in the preprocessing phase. Then in the online phase, all parties locally compute $[\boldsymbol{r}]_{t+k-1}$ from $\{[r_j|j]_t\}_{j=1}^k$ and send their shares to P_s. P_s checks whether the received shares form a valid degree-$(t+k-1)$ packed Shamir sharing. If not, all parties send their shares of $\{[r_j|j]_t\}_{j=1}^k$ to P_s and P_s helps identify a dispute pair. Otherwise, P_s distributes $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$ to all parties. Then all parties use $\mathcal{F}_{\mathsf{VerifyPub}}$ to check whether P_s distributes a valid degree-$(k-1)$ packed Shamir sharing. If not, all parties receive a new dispute pair from $\mathcal{F}_{\mathsf{VerifyPub}}$. Otherwise, all parties locally compute $[x_j|j]_t = [\boldsymbol{x}+\boldsymbol{r}]_{k-1} - [r_j|j]_t$.

The protocol Π_{Input} is described in Appendix E. The communication of handling N input gates is $\mathsf{P2P}(O(N+n))$ elements plus $O(1) \times \mathsf{BA}(O(1))$ bits accounting for total communication of expected $\mathsf{P2P}(N + n^3 \cdot \log n)$ elements in the case of success.

6.2 Output Layer

After completing the entire computation segment by segment, all parties come to the output phase. To reconstruct the output $\boldsymbol{z} \in \mathbb{F}^k$ of party P_s towards P_s, all parties send their shares of $[\boldsymbol{z}]_{t+k-1}$ to P_s. Then after checking the consistency, P_s is able to reconstruct its output.

We formally describe Π_{Output} in Appendix F. The communication is $\mathsf{P2P}(O(N+n))$ elements to handle N output gates in the case of success.

6.3 Main Protocol

Now we are ready to present our main protocol. First, we apply the deterministic circuit transformation algorithm proposed in [GPS21] to ensure the resulting circuit is friendly to the packed secret sharing technique. By doing this, we are able to pack the input gates belonging to one party in the input layer, the multiplication gates and the addition gates in each intermediate layer, and the output gates belonging to one party in the output layer into groups of gates of size k. We refer to Appendix G for more details about the requirements of the circuit transformation. In dispute control framework, we keep recording the set of active parties $\mathcal{P}$ which contains all parties initially, and the set of dispute pair of parties $\mathcal{D}$ which is empty initially. In the **preprocessing phase**, all parties invoke $\mathcal{F}_{\mathsf{RandSh}}$ and $\mathcal{F}_{\mathsf{Triples}}$ to prepare sufficient amount of random sharings of each type. In the **input phase**, all parties invoke Π_{Input} to share their groups of inputs. Before computing the circuit, we divide uniformly the whole circuit into $O(n^2)$ segments satisfying the requirements stated in Remark 1 and start to compute all the segments sequentially. Importantly, when evaluating each segment, to tackle with the fan-out gates following the computation gates in the current segment, we first handle the fan-out gates whose output wires will be used in the current segment, then divide the rest of them into several segments also satisfying the requirements in Remark 1, and compute the segments sequentially. We emphasize that without doing this, the number of fan-out operations we need to perform in one segment may be even larger than the segment size, which will result in a large communication overhead if re-evaluation occurs. We formally present our algorithm for segment division in Algorithm 1 in Appendix J.

In the **computation phase**, each segment is first evaluated by Π_{Eval}. Then all parties together check the correctness of the computation by running $\Pi_{\mathsf{VerifyMult}}$ and $\Pi_{\mathsf{VerifyShTrans}}$. All parties either agree on the success of the computation, in which case they move on to evaluate the next segment, or receive a new dispute pair. In the latter case, the whole segment is re-evaluated.

Ultimately, all parties have already had their shares of each output. Then in the **output phase**, all parties invoke Π_{Output} to reconstruct the output towards the corresponding party. We describe the main protocol Π_{Main} below.

Protocol 10 : Π_{Main} **in** $\mathcal{F}_{\mathsf{RandSh}}, \mathcal{F}_{\mathsf{Triples}}, \mathcal{F}_{\mathsf{VerifyPub}}$**-hybrid model**

Circuit Transformation Phase. We adopt the deterministic algorithm in [GPS21].
Let $\mathcal{D} = \{(P_i, P_j) | P_i \text{ and } P_j \text{ are disputed}\}$ denote the set of pairs of disputed parties which initially is an empty set. Let $\mathcal{P}$ of size n' denote the set of parties remained to participate the computation which contains all n parties initially.
Preprocessing Phase. All parties invoke $\mathcal{F}_{\mathsf{RandSh}}$, $\mathcal{F}_{\mathsf{Triples}}$ to receive correlated randomness that will be used in the online phase.
Input Phase. All parties invoke Π_{Input} to share their inputs.
Segment Division Phase. We perform Algorithm 1 in Appendix J to divide the circuit into $O(n^2)$ segments which will then be evaluated sequentially.
Computation Phase. For every segment of the circuit:

1. All parties in $\mathcal{P}$ invoke Π_{Eval} to evaluate this segment.
2. All parties in $\mathcal{P}$ invoke $\Pi_{\mathsf{VerifyMult}}$ and $\Pi_{\mathsf{VerifyShTrans}}$ to verify the correctness of the computation.

If a dispute pair of parties is identified, **re-evaluate*** the current segment. Otherwise, all parties perform the remaining fan-out gates. This can be viewed as a segment that only contains fan-out gates. Thus, it can be evaluated in the same way as described above.
Output Phase. All parties invoke Π_{Output} to reconstruct their outputs.
If a dispute pair of parties is identified, **re-evaluate*** the current segment.

***Re-evaluation**. Each time when faults occur and a dispute pair is identified, add this pair to $\mathcal{D}$, assign each dispute pair in $\mathcal{D}$ who are both active an intermediate party for relaying, and re-evaluate the current segment. (We omit this in protocol description for simplicity.)

Remark 1 (Segment Division). To upper bound the overhead of re-evaluating the segments due to the faults caused by dispute pairs, we enforce our segment division to satisfy the following three requirements: 1) the circuit size of each segment is $O(\frac{|C|}{n^2})$ 2) the circuit depth of each segment is $O(\frac{\mathsf{Depth}}{n^2})$ and 3) gates belonging to one group should be contained in the same segment. To achieve this, following the topology of the circuit, we include the gates into one segment until the circuit size of the current segment exceeds $\frac{|C|}{n^2}$ or the circuit depth of the current segment exceeds $\frac{\mathsf{Depth}}{n^2}$ while keeping the number of gates of each kind a multiple of k, which will result in at most $3n^2$ segments in the online phase.

Analysis of the Communication Complexity of Π_{Main}. Let I denote the input size, G denote the number of gates, and O denote the output size, Depth denote the depth of the circuit. Then $|C| \geq I + G + O$. We set $T = n - t, k = (n+5)/6$. In summary, the overall communication complexity for online phase is $\mathsf{P2P}(O(|C| \cdot \frac{n}{k} + (\mathsf{Depth} + n) \cdot n + n^5))$ elements plus $O(n^2) \times \mathsf{BA}(O(1))$ bits, where the term n^5 is caused by redoing the segment when encountering faults. Thus, if we apply an expected constant round broadcast protocol like [AC24], the online communication becomes expected $\mathsf{P2P}(O(|C| \cdot \frac{n}{k} + (\mathsf{Depth} + n) \cdot n + n^5 \cdot \log n))$ elements. We refer to Appendix H for more detailed analysis.

Analysis of the Round Complexity of Π_{Main}. For each segment of circuit, its depth is bounded by $O(\frac{\mathsf{Depth}}{n^2})$. The round complexity is $O(\frac{\mathsf{Depth}}{n^2}) + O(1) \times \mathsf{BA}$ and $O(\frac{\mathsf{Depth}}{n^2}) + O(1) \times \mathsf{BA} + O(1) \times \mathsf{BC}$ in the case that faults occur. Hence, the online round complexity is $O(\mathsf{Depth} + n^2) + O(n^2) \times \mathsf{BA} + O(n^2) \times \mathsf{BC}$. Thus, if we apply an expected constant round broadcast protocol like [AC24], the online round complexity becomes $O(\mathsf{Depth} + n^2)$.

6.4 Summary

In the preprocessing phase, the preprocessing randomness can be divided into two classes: (1) degree-t Shamir secret sharings satisfying some specific requirements and (2) packed Beaver triples, which both can be dealt with using the techniques in [BH08] as mentioned in Subsection 3.5. In total, the communication for preprocessing phase is $\mathsf{P2P}(O((|C| + k(n + \mathsf{Depth}))n + n^4))$ elements plus $O(n^2) \times \mathsf{BA}(O(1))$ bits, which becomes expected $\mathsf{P2P}(O((|C| + k(n + \mathsf{Depth}))n + n^5 \cdot \log n))$ elements assuming an expected constant round broadcast protocol like [AC24] is used. We refer to Appendix I for the detailed communication complexity analysis for preprocessing phase. Moreover, the round complexity for preprocessing phase is $O(n^2) + O(n^2) \times \mathsf{BA} + O(n^2) \times \mathsf{BC}$ and becomes $O(n^2)$ in expectation assuming an expected constant round broadcast protocol like [AC24] is used.

To conclude, combining the preprocessing phase and the online phase, we get a perfectly secure protocol to compute a circuit C with online communication of expected $O(|C| \cdot \frac{n}{k} + (\mathsf{Depth} + n) \cdot n + n^5 \cdot \log n)$ elements which is linear in the circuit size, offline communication of expected $O((|C| + k(n + \mathsf{Depth}))n + n^5 \cdot \log n$ elements, online round complexity $O(\mathsf{Depth} + n^2)$ in expectation and offline round complexity $O(n^2)$ in expectation.

Functionality 4 : $\mathcal{F}_{\mathsf{Main}}$ in $\mathcal{F}_{\mathsf{RandSh}}, \mathcal{F}_{\mathsf{Triples}}, \mathcal{F}_{\mathsf{VerifyPub}}$-hybrid model

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

Lemma 3. *Protocol Π_{Main} securely computes $\mathcal{F}_{\mathsf{Main}}$ in $\mathcal{F}_{\mathsf{RandSh}}$, $\mathcal{F}_{\mathsf{Triples}}$, $\mathcal{F}_{\mathsf{VerifyPub}}$-hybrid model against a fully malicious adversary who controls at most $t < n/3$ parties.*

See the proof in Appendix C.

Combining the complexity analysis, lemma 2, and lemma 3, we obtain the following theorem.

Theorem 1. *Let n denote the number of parties. Let $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq 2n$. For an arithmetic circuit C over $\mathbb{F}$, there exists an information-theoretic MPC protocol that computes C against a fully malicious adversary controlling at most $t = \frac{n-1}{3}$ corrupted parties with perfect security. The communication cost of the protocol is expected $O(|C| + \mathsf{Depth} \cdot n + n^5 \cdot \log n)$ elements for the online phase and expected $O(|C| \cdot n + \mathsf{Depth} \cdot n^2 + n^5 \cdot \log n)$ elements for the preprocessing phase, where Depth is the circuit depth. The round complexity of the protocol is $O(\mathsf{Depth} + n^2)$ in expectation for the online phase and $O(n^2)$ in expectation for the preprocessing phase.*

Acknowledgments

Y. Song was supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61361136003.

References

AAPP23. Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. Detect, pack and batch: Perfectly-secure mpc with linear communication and constant expected time. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 251–281, Cham, 2023. Springer Nature Switzerland.

AAY22. Ittai Abraham, Gilad Asharov, and Avishay Yanai. Efficient perfectly secure computation with optimal resilience. *Journal of Cryptology*, 35(4):27, Sep 2022.

AC24. Gilad Asharov and Anirudh Chandramouli. Perfect (parallel) broadcast in constant expected rounds via statistical vss. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 310–339, Cham, 2024. Springer Nature Switzerland.

ALR11. Gilad Asharov, Yehuda Lindell, and Tal Rabin. Perfectly-secure multiplication for any t < n/3. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 240–258, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

Bea89. Donald Beaver. Multiparty protocols tolerating half faulty processors. In *Conference on the Theory and Application of Cryptology*, pages 560–572. Springer, 1989.

BGW88. Ben-Or, Michael, 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.

BH06. Zuzana Beerliová-Trubíniová and Martin Hirt. Efficient multi-party computation with dispute control. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 305–328, New York, NY, USA, March 4–7, 2006. Springer, Heidelberg, Germany.

BH08. Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008: 5th Theory of Cryptography Conference*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230, San Francisco, CA, USA, March 19–21, 2008. Springer, Heidelberg, Germany.

Can00. Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13:143–202, 2000.

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.

DEN24. Anders Dalskov, Daniel Escudero, and Ariel Nof. Fully secure mpc and zk-fliop over rings: New constructions, improvements and extensions. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024*, pages 136–169, Cham, 2024. Springer Nature Switzerland.

DIK10. Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 445–465, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

DN07. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany.

DS20. Ivan Damgård and Nikolaj I. Schwartzbach. Communication lower bounds for perfect maliciously secure MPC. Cryptology ePrint Archive, Paper 2020/251, 2020.

EGPS22. Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. TurboPack: Honest majority MPC with constant online communication. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 951–964, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

FY92. Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *24th Annual ACM Symposium on Theory of Computing*, pages 699–710, Victoria, BC, Canada, May 4–6, 1992. ACM Press.

GBO+23. Mariana Gama, Emad Heydari Beni, Emmanuela Orsini, Nigel P. Smart, and Oliver Zajonc. Mpc with delayed parties over star-like networks. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023*, pages 172–203, Singapore, 2023. Springer Nature Singapore.

GLO+21. Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. Atlas: Efficient and scalable mpc in the honest majority setting. In *Advances in Cryptology – CRYPTO 2021*, pages 244–274, Cham, 2021. Springer International Publishing.

GLS19. Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 85–114, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.

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.

GPS21. Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 275–304, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.

GPS22. Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority MPC with packed secret sharing. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 3–32, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany.

GSZ20. Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority mpc. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 618–646, Cham, 2020. Springer International Publishing.

HMP00. Martin Hirt, Ueli Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, pages 143–161, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

RB89. Tal Rabin and Ben-Or, Michael. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85. ACM, 1989.

Sha79. Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 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 Proof of Lemma 1

Proof. Suppose for all $1 \le h \le T$, P_j's share for U_h is $u_h^{(j)}, \forall 1 \le j \le n$. Then in step 2, for all $1 \le i \le T+t$, every party P_j computes its share for $U(\beta_i)$ as $u^{(j)}(\beta_i) = \sum_{h=1}^{T} u_h^{(j)} \beta_i^{h-1}$. Define $\mathbf{V} = \begin{bmatrix} 1 & \beta_1 & \dots & \beta_1^{T-1} \\ 1 & \beta_2 & \dots & \beta_2^{T-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \beta_{n'} & \dots & \beta_{n'}^{T-1} \end{bmatrix}$, which is the Vandermonde matrix of size $n' \times T$. Then we get $(u^{(j)}(\beta_1), \dots, u^{(j)}(\beta_{n'}))^{\mathrm{T}} = \mathbf{V} \cdot (u_1^{(j)}, \dots, u_T^{(j)})^{\mathrm{T}}, \forall 1 \le j \le n'$. Suppose the set of honest parties with smallest T indexes in $\{1, 2, \dots, n'\}$ by $\mathcal{H}'$. Then we further get $(u^{(j)}(\beta_i))_{i \in \mathcal{H}'}^{\mathrm{T}} = \mathbf{V}_{\mathcal{H}'} \cdot (u_1^{(j)}, \dots, u_T^{(j)})^{\mathrm{T}}, \forall 1 \le j \le n'$, where $\mathbf{V}_{\mathcal{H}'}$ denotes the matrix consisting of the rows in $\mathbf{V}$ with their indexes in $\mathcal{H}'$. Now suppose all parties in $\mathcal{H}'$ get `happy`, which implies for all $i \in \mathcal{H}'$, $(u^{(1)}(\beta_i), \dots, u^{(n')}(\beta_i))$ satisfy the linear secret sharing scheme. Since the Vandermonde matrix $\mathbf{V}$ is super-invertible, which means any its $T \times T$ submatrix is invertible, then $\mathbf{V}_{\mathcal{H}'}$ is invertible and this is a bijective map. Hence, every sharing U_h with P_j's share to be $u_h^{(j)}$ must be consistent as $(u_1^{(j)}, \dots, u_T^{(j)})^{\mathrm{T}} = \mathbf{V}_{\mathcal{H}'}^{-1} \cdot (u^{(j)}(\beta_i))_{i \in \mathcal{H}'}^{\mathrm{T}}, \forall 1 \le j \le n'$, which leads to a contradiction. □

B Proof of Lemma 2

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 $\Pi_{\mathsf{VerifyPub}}$. Now we describe the construction of the simulator $\mathcal{S}$.

- $\mathcal{S}$ invokes $\mathcal{F}_{\mathsf{VerifyPub}}$ with the set of updated dispute set $\mathcal{D}$, and receives honest parties' shares of N linear Σ-sharings from $\mathcal{F}_{\mathsf{VerifyPub}}$. For every T Σ-sharings $\{U_1, \dots, U_T\}$, $\mathcal{S}$ simulates as follows:
  - In step 1, for every honest party, $\mathcal{S}$ follows the protocol honestly, (i.e. $\mathcal{S}$ computes $U(\beta_j) = \sum_{h=1}^{T} U_h \beta_j^{h-1}$) and sends its shares of $\{U(\beta_j)\}_{P_j \in \mathcal{C}}$ to the adversary.
    For every corrupted party, $\mathcal{S}$ receives from the adversary its shares of $\{U(\beta_j)\}_{P_j \in \mathcal{H}}$.
  - In step 2, for every honest party P_i, $\mathcal{S}$ checks whether their shares of $U(\beta_i)$ and the shares received from the adversary form a consistent Σ-sharing.
    - If the check fails, $\mathcal{S}$ records (P_i,`unhappy`).
  - In step 3, for every honest party P_i, if $\mathcal{S}$ has recorded (P_i,`unhappy`), $\mathcal{S}$ sends `unhappy` to the adversary and records `reject` for the current iteration. Otherwise, $\mathcal{S}$ sends `happy` to the adversary.
    For corrupted parties, if $\mathcal{S}$ receives `unhappy` from the adversary, it records `reject` for the current iteration.
    If $\mathcal{S}$ does not neither record any (P_i, `unhappy`) for some honest party P_i nor receive `unhappy` from the adversary, it records `accept` for current iteration, and continues for another iteration or ends loop if all N Σ-sharings have been checked.
  - In step 4, if $\mathcal{S}$ records `reject` for the current iteration, it simulates the fault-localization step. We consider two situations below:
    - If P_r is honest, $\mathcal{S}$ follows the protocol honestly and finds a new dispute pair on behalf of honest P_r.
    - If P_r is corrupted, since $\mathcal{S}$ knows all the information about honest parties, $\mathcal{S}$ is able to act as honest parties, follow the protocol honestly, and records the dispute pair broadcast by P_r.

  $\mathcal{S}$ ends loop.
- If $\mathcal{S}$ records `accept` for all iterations, $\mathcal{S}$ sends `accept` to $\mathcal{F}_{\mathsf{VerifyPub}}$. Otherwise, $\mathcal{S}$ sends `reject` to $\mathcal{F}_{\mathsf{VerifyPub}}$ together with the recording dispute pair.
- $\mathcal{S}$ outputs what the adversary $\mathcal{A}$ outputs.

Now we show $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Note that $\mathcal{S}$ knows all the information about honest parties from $\mathcal{F}_{\mathsf{VerifyPub}}$ and thus can simulate all the messages sent by honest parties. Hence, here we only need to show that: in the real world, following the protocol,

1. when all parties get `happy`, the shares of honest parties must be consistent;
2. if at least party gets `unhappy`, the fault-localization step in the protocol can always find a new pair of disputed parties containing at least one corrupted party.

For 1., by lemma 1, at least one honest party will get `unhappy` if the shares of U_h belonging to honest parties are inconsistent for some $1 \leq h \leq T$. Since all honest parties get `happy`, the shares of honest parties are consistent. For 2., we discuss this in two situations below.

- If a corrupted party P_r is `unhappy`, then it is expected to broadcast a new pair of parties or it will be believed to be corrupted as in the first case in 4(e). We only need to prove it cannot succeed in outputting a pair of two honest parties. This is clear because considering in step 4(e), two adjacent honest parties will not agree with different messages.
- If an honest party P_r is `unhappy`, we show that P_r is always able to figure out a new pair of dispute parties containing at least one corrupted party. Since P_r is `unhappy`, then 1) the shares of $\{U_h\}_{1\leq h\leq T}$ belonging to honest parties are inconsistent or(and) 2) a corrupted party sends an incorrect share of $U(\beta_r)$ or(and) an incorrect happy-bit to P_r. In both cases, we have a) there exists a pair of parties (P_i, P_j) such that the messages claimed by P_i sending to P_j and by P_j receiving from P_i are inconsistent or(and) b) P_j is observed not following the protocol when doing local computation. Notice a) can be solved by the third case in 4(e) and b) can be solved by the second case in 4(e).

Now we complete the proof. □

C Proof of Lemma 3

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 Π_{Main}. We now describe the construction of the simulator $\mathcal{S}$. When simulating $\mathcal{F}_{\mathsf{VerifyPub}}$, for simplicity, we omit to let $\mathcal{S}$ send the set of dispute pairs $\mathcal{D}$ to $\mathcal{F}_{\mathsf{VerifyPub}}$.

Preprocessing phase. $\mathcal{S}$ emulates the ideal functionalities $\mathcal{F}_{\mathsf{RandSh}}$ and $\mathcal{F}_{\mathsf{Triples}}$, and receives the shares of corrupted parties for each secret sharing. Note that $\mathcal{F}_{\mathsf{RandSh}}$ and $\mathcal{F}_{\mathsf{Triples}}$ do not need to send any message to corrupted parties.

Input phase. $\mathcal{S}$ simulates Π_{Input} as follows. For each group of k input gates belonging to party P_s with input $\boldsymbol{x} = (x_1, \ldots, x_k) \in \mathbb{F}^k$:

1. For step 1 - 4,
   - If P_s is honest, $\mathcal{S}$ checks whether the receiving shares $[\boldsymbol{r}]_{t+k-1}$ from all parties form a degree-$(t+k-1)$ sharing honestly. If the check is passed, $\mathcal{S}$ samples k random field elements as $\boldsymbol{x}+\boldsymbol{r}$, computes $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$, and sends the corrupted parties' shares of $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$ to the adversary $\mathcal{A}$.
   - If P_s is corrupted, $\mathcal{S}$ samples k random field elements as $\boldsymbol{r}$. Based on the secret $\boldsymbol{r}$ and the corrupted parties' shares, $\mathcal{S}$ samples the whole sharing $[\boldsymbol{r}]_{t+k-1}$ and sends the shares of $[\boldsymbol{r}]_{t+k-1}$ held by honest parties to the adversary $\mathcal{A}$.
     - If P_s broadcasts the check fails, then for $j \in [k]$, $\mathcal{S}$ computes honest parties' shares of $[r_j|j]_t$ based on the sampled secret r_j and corrupted parties' shares of $[r_j|j]_t$, and sends them to P_s on behalf of honest parties.

     If P_s broadcasts the check is passed, then $\mathcal{S}$ receives honest parties' shares of $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$, reconstructs $\boldsymbol{x}+\boldsymbol{r}$, computes and records $\boldsymbol{x}$.
2. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on honest parties' shares of $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$ and the adversary $\mathcal{A}$'s instruction. $\mathcal{S}$ computes and records the corrupted parties' shares of $[x_j|j]_t = [\boldsymbol{x}+\boldsymbol{r}]_{k-1} - [r_j|j]_t, \forall j \in [k]$.

Computation phase.

Simulating $\Pi_{\mathsf{zeroSharing}}$.

1. For step 1, for each honest party P_s, $\mathcal{S}$ samples a random element for each corrupted party as its share of $[\mathbf{0}^{(s)}]_{n'-1}$. For each corrupted party P_s, $\mathcal{S}$ receives the shares of $[\mathbf{0}^{(s)}]_{n'-1}$ held by honest parties from the adversary $\mathcal{A}$.

Simulating $\Pi_{\mathsf{Transpose}}$. If P_{king} is corrupted,

1. For step 1 and 2, $\mathcal{S}$ knows corrupted parties' shares of $\{[x_{i,j}+r_{i,j}|j]_t\}_{i,j=1}^k$. $\mathcal{S}$ further samples random field elements as $\{x_{i,j}+r_{i,j}\}_{i,j=1}^k$ to compute the honest parties' shares of $\{[x_{i,j}+r_{i,j}|j]_t\}_{i,j=1}^k$ and $\{[\boldsymbol{x}_i+\boldsymbol{r}_i]_{t+k-1}\}_{i=1}^k$, and sends the second part to P_{king} on behalf of the honest parties.
2. For step 3, if P_{king} broadcasts the i-th sharing is inconsistent, $\mathcal{S}$ sends the honest parties' shares of $\{[x_{i,j}+r_{i,j}|j]_t\}_{j=1}^k$ to P_{king}.
3. For step 4, $\mathcal{S}$ records the receiving shares of $\{[\boldsymbol{x}_{*,i}+\boldsymbol{r}_{*,i}]_{k-1}\}_{i=1}^k$ held by honest parties. Then $\mathcal{S}$ computes the corrupted parties' shares of $\{[\boldsymbol{x}_{*,i}+\boldsymbol{r}_{*,i}]_{k-1}\}_{i=1}^k$.
4. For step 6, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the honest parties' shares received from P_{king} and the adversary $\mathcal{A}$'s instruction.

If P_{king} is honest,

1. For step 1, for $i \in [k]$, $\mathcal{S}$ computes the corrupted parties' shares of $[\boldsymbol{x}_i]_{t+k-1}$ and $[\boldsymbol{r}_i]_{t+k-1}$.
2. For step 2, for $i \in [k]$, $\mathcal{S}$ receives the corrupted parties' shares of $[\boldsymbol{x}_i+\boldsymbol{r}_i]_{t+k-1}$ from the adversary $\mathcal{A}$.
3. For step 3, $\mathcal{S}$ checks whether the computing shares correspond to the receiving shares. If they are inconsistent, $\mathcal{S}$ broadcasts a complain, receives the corrupted parties' shares, and figures out a new dispute pair.
4. For step 4, $\mathcal{S}$ samples random field elements as $\{x_{i,j}+r_{i,j}\}_{i,j=1}^k$ and distributes the corrupted parties' shares of $\{[\boldsymbol{x}_{*,i}+\boldsymbol{r}_{*,i}]_{k-1}\}_{i=1}^k$ to the adversary $\mathcal{A}$.
5. For step 5, $\mathcal{S}$ computes and records the corrupted parties' shares of $\{[x_{j,i}|j]_t\}_{i,j=1}^k$.
6. For step 6, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the adversary $\mathcal{A}$'s instruction.

Simulating $\Pi_{\mathsf{PrepTrans}}$.

1. For step 1, $\mathcal{S}$ computes and records the corrupted parties' shares of $\{[L_{j,i}(\boldsymbol{r}_j)|j]_t\}_{i,j=1}^k$.
2. For step 2, $\mathcal{S}$ simulates $\Pi_{\mathsf{Transpose}}$ as described above.
3. For the last step of verifying the "transpose" operations in a batch-wise manner, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the honest parties' shares received from P_{king} and the adversary $\mathcal{A}$'s instruction. If the check has been passed, $\mathcal{S}$ computes and records the corrupted parties' shares of $\{[L_{i,j}(\boldsymbol{r}_i)|j]_t\}_{i,j=1}^k$.

Simulating Π_{Mult}. If P_{king} is corrupted,

1. For step 2, for each honest party P_i, $\mathcal{S}$ samples 2 random elements as P_i's shares of $[\boldsymbol{x}+\boldsymbol{a}]_{n'-1}, [\boldsymbol{y}+\boldsymbol{b}]_{n'-1}$, and sends to P_{king} on behalf of P_i.
2. For step 3, $\mathcal{S}$ receives honest parties' shares of $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{k-1}$ from P_{king} and records them for verification.
3. For step 4, for each honest party, $\mathcal{S}$ samples a random field element as its share of $[\boldsymbol{z}+\boldsymbol{r}]_{n'-1}$ and sends it to P_{king}.
4. For step 5, $\mathcal{S}$ receives honest parties' shares of $[\boldsymbol{z}+\boldsymbol{r}]_{k-1}$ and records them used for verification.

If P_{king} is honest,

1. For step 2, $\mathcal{S}$ receives corrupted parties' shares of $[\boldsymbol{x}+\boldsymbol{a}]_{n'-1}, [\boldsymbol{y}+\boldsymbol{b}]_{n'-1}$ from the adversary.
2. For step 3, $\mathcal{S}$ samples $2k$ random elements as the secrets $\boldsymbol{x}+\boldsymbol{a}, \boldsymbol{y}+\boldsymbol{b}$, computes the corrupted parties' shares of $[\boldsymbol{x}+\boldsymbol{a}]_{k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{k-1}$, and sends them to the adversary on behalf of the honest party P_{king}.
3. For step 4, $\mathcal{S}$ receives the corrupted parties' shares of $[\boldsymbol{z}+\boldsymbol{r}]_{n'-1}$ from the adversary and records them used for verification.
4. For step 5, $\mathcal{S}$ samples k random elements as the secret $\boldsymbol{z}+\boldsymbol{r}$, computes the corrupted parties' shares of $[\boldsymbol{z}+\boldsymbol{r}]_{k-1}$, and sends them to the adversary on behalf of honest party P_{king}.
5. For step 6, $\mathcal{S}$ computes and records the corrupted parties' shares of $[z_j|j]_t$.

Simulating Π_{ShTrans}. If P_{king} is corrupted,

1. For step 2, for each honest party, $\mathcal{S}$ samples a random element as P_s's share of $[\boldsymbol{x}+\boldsymbol{r}]_{n'-1}$ and sends to P_{king} on behalf of the honest party.
2. For step 3, $\mathcal{S}$ receives honest parties' shares of $[L_(\boldsymbol{x}+\boldsymbol{r})]_{k-1}$ from the adversary and records them for the purpose of verification.

If P_{king} is honest,

1. For step 2, $\mathcal{S}$ receives corrupted parties' shares of $[\boldsymbol{x}+\boldsymbol{r}]_{n'-1}$ from the adversary $\mathcal{A}$.
2. For step 3, $\mathcal{S}$ samples k random elements as the secret $\boldsymbol{x}+\boldsymbol{r}$, computes corrupted parties' shares of $[L(\boldsymbol{x}+\boldsymbol{r})]_{k-1}$, and sends them to the adversary on behalf of honest party P_{king}.
3. For step 4, $\mathcal{S}$ computes and records the corrupted parties' shares of $\{[L_i(\boldsymbol{x}+\boldsymbol{r})|i]_t\}_{i=1}^k$.

Simulating $\Pi_{\mathsf{VerifyMult}}$. If P_{king} is corrupted,

1. For step 1, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the honest parties' shares received from P_{king} and the adversary $\mathcal{A}$'s instruction.
   If the verification has been passed, $\mathcal{S}$ is able to reconstruct $\boldsymbol{x}+\boldsymbol{a}, \boldsymbol{y}+\boldsymbol{b}, \boldsymbol{z}+\boldsymbol{r}$. Together with corrupted parties' shares of $[\boldsymbol{x}+\boldsymbol{a}]_{t+k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{t+k-1}, [\boldsymbol{z}+\boldsymbol{r}]_{t+2k-2}$, $\mathcal{S}$ computes honest parties' shares of $[\boldsymbol{x}+\boldsymbol{a}]_{t+k-1}, [\boldsymbol{y}+\boldsymbol{b}]_{t+k-1}, [\boldsymbol{z}+\boldsymbol{r}]_{t+2k-2}$, and computes honest parties' shares of $[\mathbf{0}_1]_{n'-1}, [\mathbf{0}_2]_{n'-1}, [\mathbf{0}_3]_{n'-1}$ by $[\mathbf{0}_1]_{n'-1} = [\boldsymbol{x}+\boldsymbol{a}]_{n'-1} - [\boldsymbol{x}+\boldsymbol{a}]_{t+k-1}, [\mathbf{0}_2]_{n'-1} = [\boldsymbol{y}+\boldsymbol{b}]_{n'-1} - [\boldsymbol{y}+\boldsymbol{b}]_{t+k-1}, [\mathbf{0}_3]_{n'-1} = [\boldsymbol{z}+\boldsymbol{r}]_{n'-1} - [\boldsymbol{z}+\boldsymbol{r}]_{t+2k-2}$. $\mathcal{S}$ further computes the sharings distributed by the honest parties in $\Pi_{\mathsf{zeroSharing}}$ by inverting the corresponding submatrix of the Vandermonde matrix and sends them together with honest parties' shares of $[\mathbf{0}_1]_{n'-1}, [\mathbf{0}_2]_{n'-1}, [\mathbf{0}_3]_{n'-1}$ to P_{king} on behalf of honest parties.
2. For step 2,
   - If P_{king} broadcasts the check is passed, then go to the next step.
   - If P_{king} broadcasts the check fails, $\mathcal{S}$ computes honest parties' shares of $\{[x_j+a_j|j]_t, [y_j+b_j|j]_t\}_{j=1}^k$ based on the sampled secrets $\boldsymbol{x}+\boldsymbol{a}, \boldsymbol{y}+\boldsymbol{b}$ and corrupted parties' shares of $\{[x_j+a_j|j]_t, [y_j+b_j|j]_t\}_{j=1}^k$ and sends to P_{king} on behalf of honest parties.
3. For step 3, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the honest parties' shares sent to P_{king} and received from P_{king} and the adversary $\mathcal{A}$'s instruction.
4. For step 4,
   - if P_{king} broadcasts the check is passed, then go to the next step.
   - if P_{king} broadcasts the check fails.
     - For step 4(a), $\mathcal{S}$ samples random elements as the secrets $\boldsymbol{v}$, computes honest parties' shares of $[\boldsymbol{v}]_{t+2k-2}$ based on the sampled secrets and the corrupted parties' shares, computes honest parties' shares of $[\boldsymbol{v}]_{t+2k-2}$ based on the sampled secrets and the corrupted parties' shares, and sends them to P_{king} on behalf of honest parties.
     - For step 4(b), if P_{king} broadcasts the check fails, then $\mathcal{S}$ samples $\boldsymbol{a}', \boldsymbol{b}', \boldsymbol{c}'$ and computes the honest parties' shares of $\{[a'_j|j]_t, [b'_j|j]_t, [c'_j|j]_t\}_{j=1}^k, \{[r'_j|j]_t\}_{j=1}^{2k-1}$, and sends them to P_{king} on behalf of honest parties.
     - For step 4(c), if P_{king} broadcasts the check is passed, then $\mathcal{S}$ samples random elements as the secrets $\boldsymbol{a}'+\boldsymbol{a}, \boldsymbol{b}'+\boldsymbol{b}, \boldsymbol{c}'+\boldsymbol{c}$, computes the honest parties' shares of $\{[a'_j+a_j|j]_t, [b'_j+b_j|j]_t, [c'_j+c_j|j]_t\}_{j=1}^k, \{[r'_j+r_j|j]_t\}_{j=1}^{2k-1}$ based on the sampled secrets and the corrupted parties' shares, and sends them to P_{king} on behalf of honest parties.
5. For step 5, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the honest parties' shares sent to and received from P_{king} and the adversary $\mathcal{A}$'s instruction. If the check has been passed, $\mathcal{S}$ computes and records the corrupted parties' shares of $\{[z_j|j]_t\}_{j=1}^k$.

If P_{king} is honest,

1. For step 1, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the adversary $\mathcal{A}$'s instruction.

2. For step 2, $\mathcal{S}$ receives corrupted parties' shares of $[\mathbf{0}_1]_{n'-1}, [\mathbf{0}_2]_{n'-1}$ in step 1 and checks if they satisfy $[\mathbf{0}_1]_{n'-1} = [\boldsymbol{x}+\boldsymbol{a}]_{n'-1} - [\boldsymbol{x}+\boldsymbol{a}]_{t+k-1}, [\mathbf{0}_2]_{n'-1} = [\boldsymbol{y}+\boldsymbol{b}]_{n'-1} - [\boldsymbol{y}+\boldsymbol{b}]_{t+k-1}$. If they are inconsistent, $\mathcal{S}$ follows the protocol.
3. For step 3, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the adversary $\mathcal{A}$'s instruction.
4. For step 4, $\mathcal{S}$ checks whether the receiving corrupted parties' shares satisfy $[\mathbf{0}_3]_{n'-1} = [\boldsymbol{z}+\boldsymbol{r}]_{n'-1} - [\boldsymbol{z}+\boldsymbol{r}]_{t+2k-2}$. If they are inconsistent,
   - $\mathcal{S}$ receives the corrupted parties' shares of $[\boldsymbol{v}]_{t+2k-2}$ from the adversary $\mathcal{A}$.
   - $\mathcal{S}$ checks whether the adversary $\mathcal{A}$ sends the shares of $[\boldsymbol{v}]_{t+2k-2}$ correctly. If they are incorrect, $\mathcal{S}$ further checks whether the adversary $\mathcal{A}$ sends the corrupted parties' shares of $\{[a'_j|j]_t, [b'_j|j]_t, [c'_j|j]_t\}_{j=1}^{k}$ and $\{[r'_j|j]_t\}_{j=1}^{2k-1}$ correctly.
   - If the receiving shares in 4(a) are correct, $\mathcal{S}$ checks whether the adversary $\mathcal{A}$ sends the corrupted parties' shares of $\{[a'_j + a_j|j]_t, [b'_j + b_j|j]_t, [c'_j + c_j|j]_t\}_{j=1}^{k}, \{[r'_j + r_j|j]_t\}_{j=1}^{2k-1}$ correctly.
5. For step 5, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the adversary $\mathcal{A}$'s instruction.

Simulating $\Pi_{\mathsf{VerifyShTrans}}$. If P_{king} is corrupted,

1. For step 1, $\mathcal{S}$ simulates as it does when simulating $\Pi_{\mathsf{VerifyMult}}$ and thus obtains the honest parties' shares of $\{[\boldsymbol{x}_i + \boldsymbol{r}_i]_{t+k-1}\}_{i\in[k]}$.
2. For step 2,
   - for step 2(a), $\mathcal{S}$ receives the honest parties' shares of $\{[\boldsymbol{u}_{*,i}]_{k-1}, [\boldsymbol{v}_{*,i}]_{k-1}\}_{i=1}^{k}$ from the adversary $\mathcal{A}$.
   - for step 2(b), $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the honest parties' shares sent to and received from P_{king} and the adversary $\mathcal{A}$'s instruction.
   - for step 2(c), $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the honest parties' shares sent to and received from P_{king} and the adversary $\mathcal{A}$'s instruction.
   - for step 2(d) and 2(e), $\mathcal{S}$ computes the honest parties' shares of $\{[\boldsymbol{o}_i]_{2k-1}\}_{i\in[k]}$ and emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based on the honest parties' shares (effectively) received from P_{king} and the adversary $\mathcal{A}$'s instruction. If the check has been passed, $\mathcal{S}$ computes and records the corrupted parties' shares of $\{[L_{i,j}(\boldsymbol{x}_i)|j]_t\}_{i,j=1}^{k}$.

If P_{king} is honest,

1. For step 1, $\mathcal{S}$ simulates as it does when simulating $\Pi_{\mathsf{VerifyMult}}$.
2. For step 2,
   - for step 2(a), $\mathcal{S}$ follows the protocol and sends the corrupted parties' shares of $\{[\boldsymbol{u}_{*,i}]_{k-1}, [\boldsymbol{v}_{*,\boldsymbol{i}}]_{k-1}\}_{i=1}^{k}$ to the adversary $\mathcal{A}$ on behalf of P_{king}.
   - for step 2(b), $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based the adversary $\mathcal{A}$'s instruction.
   - for step 2(c), $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based the adversary $\mathcal{A}$'s instruction.
   - for step 2(e), $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{VerifyPub}}$ based the adversary $\mathcal{A}$'s instruction.

Output Phase. $\mathcal{S}$ simulates for Π_{Output} as follows. For each group of k output gates belonging to party P_s with output $\boldsymbol{z} = (z_1, \ldots, z_k) \in \mathbb{F}^k$. $\mathcal{S}$ together with honest parties invoke $\mathcal{F}_{\mathsf{Main}}$ with the corrupted parties' inputs computed in input phase and honest parties' inputs, respectively.

- If P_s is honest, $\mathcal{S}$ receives the corrupted parties' shares of $[\boldsymbol{z}]_{t+k-1}$ from the adversary $\mathcal{A}$. $\mathcal{S}$ checks the correctness of these shares by comparing them with the corrupted parties' shares of $[\boldsymbol{z}]_{t+k-1}$ recorded previously. If they are inconsistent, $\mathcal{S}$ figures out a new dispute pair.
- If P_s is corrupted, $\mathcal{S}$ learns P_s' output $\boldsymbol{z}$ from $\mathcal{F}_{\mathsf{Main}}$. Based on corrupted parties' shares of $\{[z_j|j]_t\}_{j=1}^{k}$, $\mathcal{S}$ computes the whole sharings of $[\boldsymbol{z}]_{t+k-1}$ and $\{[z_j|j]_t\}_{j=1}^{k}$, and sends the honest parties' shares of $[\boldsymbol{z}]_{t+k-1}$ on behalf of the honest parties.
  - If P_s broadcasts the check is passed, go to the simulation of the next segment.
  - If P_s broadcasts the check fails, $\mathcal{S}$ sends the honest parties' shares of $\{[z_j|j]_t\}_{j=1}^{k}$ computed previously to P_s on behalf of the honest parties.

$\mathcal{S}$ outputs what the adversary $\mathcal{A}$ outputs and the honest parties output their results received from $\mathcal{F}_{\mathsf{Main}}$.

Now we complete the description of the simulation. We stress that along the entire simulation, starting from receiving corrupted parties' shares of random sharings from $\mathcal{F}_{\mathsf{RandSh}}$ and $\mathcal{F}_{\mathsf{Triples}}$, $\mathcal{S}$ always computes and records the corrupted parties' shares of each secret sharing. Furthermore, $\mathcal{S}$ always makes sure the adversary cannot deviate from the protocol when encountering verification.

Above, we only simulate for a fully malicious adversary controlling exactly t corrupted parties (in the beginning). However, for a malicious adversary controlling a set of corrupted parties of size less than t, we just enroll an honest party P_i, generate random shares as corrupted parties' shares of P_i's input sharing $\{[x_j|j]_t\}_{j=1}^k$. We also generate random elements as P_i's shares of preprocessing random sharings. Thus, this will not leak P_i's information. During the simulation, we just treat P_i as a corrupted party. Hence, we can always assume there are exactly t corrupted parties when simulating.

We show that $\mathcal{S}$ perfectly simulates honest parties, $\mathcal{F}_{\mathsf{RandSh}}$, $\mathcal{F}_{\mathsf{Triples}}$ and $\mathcal{F}_{\mathsf{VerifyPub}}$' behaviors. It is sufficient to focus on the places where honest parties and functionalities need to communicate with corrupted parties.

- In the input phase, for each group of k input gates belonging to P_s, if P_s is honest, $\mathcal{S}$ needs to simulate the check and corrupted parties' shares of $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$ sent by P_s. In the ideal world, $\mathcal{S}$ simply checks whether the adversary sends their shares of $[\boldsymbol{r}]_{t+k-1}$ the same as the shares the adversary sends to $\mathcal{S}$ when emulating $\mathcal{F}_{\mathsf{RandSh}}$ and samples k random elements as $\boldsymbol{x}+\boldsymbol{r}$. Since $\boldsymbol{r}$ is uniformly random given corrupted parties' shares of $\{[r_j|j]_t\}_{j=1}^k$, $\boldsymbol{x}+\boldsymbol{r}$ is uniformly random. Therefore the distribution of $[\boldsymbol{x}+\boldsymbol{r}]_{k-1}$ simulated by $\mathcal{S}$ has the same distribution as that in the real world.
  If P_s is corrupted, $\mathcal{S}$ needs to simulate honest parties' shares of $[\boldsymbol{r}]_{t+k-1}$ as well as $\{[r_j|j]_t\}_{j=1}^k$. $\boldsymbol{r}$ is uniformly random given corrupted parties' shares of $\{[r_j|j]_t\}_{j=1}^k$ by the property of Shamir secret sharing scheme. In the ideal world, $\mathcal{S}$ randomly samples $\boldsymbol{r} \in \mathbb{F}^k$ and then computes honest parties' shares of $[\boldsymbol{r}]_{t+k-1}$. Therefore, the distribution of honest parties' shares of $[\boldsymbol{r}]_{t+k-1}$ is identical to that in the real world. Moreover, if the check has been passed, $\mathcal{S}$ is able to compute P_s's input $\boldsymbol{x}$ and will feed it to $\mathcal{F}_{\mathsf{Main}}$ to get outputs.
  Note in both case, $\mathcal{S}$ is able to compute corrupted parties' shares of $\{[x_j|j]_t\}_{j=1}^k$ and ensure that the adversary follows the protocol.
- In the computation phase, we will show that $\mathcal{S}$ can always learn corrupted parties' shares of $\{[z_j^G|j]_t\}_{j=1}^k$ for each group G of k wires in the circuit. Furthermore, after verification, corrupted parties' shares of all wires $\{\{[z_j^G|j]_t\}_{j=1}^k\}_G$ maintained by $\mathcal{S}$ have the same joint distribution as that in the real world. Note that this is true for the first layer (the input layer).

  - Preparing random $\mathbf{0}$-sharings (simulating $\Pi_{\mathsf{zeroSharing}}$). $\mathcal{S}$ receives the honest parties' shares of $\mathbf{0}$-sharings distributed by the adversary $\mathcal{A}$ and sends random field elements as the corrupted parties' shares of $\mathbf{0}$-sharings distributed by honest parties, which does not influence the output distribution.
  - Preparing random sharings for sharing transformation (simulating $\Pi_{\mathsf{Transpose}}$ and $\Pi_{\mathsf{PrepTrans}}$). If P_{king} is corrupted, $\mathcal{S}$ samples random elements as $\{\boldsymbol{x}_i+\boldsymbol{r}_i\}_{i=1}^k$ and computes the honest parties shares of $\{[\boldsymbol{x}_i+\boldsymbol{r}_i]_{t+k-1}\}_{i=1}^k$ and $\{[x_{i,j}+r_{i,j}|j]_t\}_{i,j=1}^k$ if in need, which does not influence the joint distribution of honest parties' outputs and corrupted parties' shares of $\{[x_{j,i}|j]_t\}_{i,j=1}^k$ since $\boldsymbol{r}$ is uniformly random given corrupted parties' shares of $\{[r_{i,j}|j]_t\}_{i,j=1}^k$ and the adversary $\mathcal{A}$ is forced to distribute degree-$(k-1)$ sharings $\{[\boldsymbol{x}_{*,i}+\boldsymbol{r}_{*,i}]_{k-1}\}_{i=1}^k$ correctly.
    If P_{king} is honest, $\mathcal{S}$ samples random field elements as $\{\boldsymbol{x}_i+\boldsymbol{r}_i\}_{i=1}^k$, which does not influence the joint distribution of honest parties' outputs and corrupted parties' shares of $\{[x_{j,i}|j]_t\}_{i,j=1}^k$ since $\{\boldsymbol{x_i}+\boldsymbol{r_i}\}_{i=1}^k$ are uniformly random given corrupted parties' shares of $\{[x_{i,j}+r_{i,j}|j]_t\}_{i,j=1}^k$ and the adversary $\mathcal{A}$ is forced to send the corrupted parties' shares of $\{[\boldsymbol{x}_i+\boldsymbol{r}_i]_{t+k-1}\}_{i=1}^k$ correctly.
    Note that in both cases, $\mathcal{S}$ is able to compute corrupted parties' shares of $\{[x_{j,i}|j]_t\}_{i,j=1}^k$ if the verification has been passed.
  - Addition gates. $\mathcal{S}$ simply computes the corrupted parties' shares by local addition and records the results.
  - Multiplication gates (simulating Π_{Mult} and $\Pi_{\mathsf{VerifyMult}}$). If P_{king} is corrupted, when simulating Π_{Mult}, $\mathcal{S}$ samples random elements as honest parties' shares of $[\boldsymbol{x}+\boldsymbol{a}]_{n'-1}$, $[\boldsymbol{y}+\boldsymbol{b}]_{n'-1}$ and $[\boldsymbol{z}+\boldsymbol{r}]_{n'-1}$, which

does not change the distribution of corrupted parties' shares of $\{[z_j|j]_t\}_{j=1}^k$ since honest parties' shares of $[\mathbf{0}_1]_{n'-1}, [\mathbf{0}_2]_{n'-1}, [\mathbf{0}_3]_{n'-1}$ are uniformly random and $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{r}$ are uniformly random in the view of the adversary $\mathcal{A}$. When simulating $\Pi_{\mathsf{VerifyMult}}$, $\mathcal{S}$ samples random elements as the secret $\boldsymbol{v}$ and samples random elements as the secrets $(\boldsymbol{a'}, \boldsymbol{b'}, \boldsymbol{c'}, \boldsymbol{r'})$ or $(\boldsymbol{a'}+\boldsymbol{a}, \boldsymbol{b'}+\boldsymbol{b}, \boldsymbol{c'}+\boldsymbol{c}, \boldsymbol{r'}+\boldsymbol{r})$, which does not change the joint distribution of honest parties' outputs and corrupted parties' shares of $\{[z_j|j]_t\}_{j=1}^k$ due to the property of Shamir secret sharing scheme that the secrets are uniformly random given the corrupted parties' shares. The distribution will coincide with that in real world if the verification is passed.

If P_{king} is honest, when simulating Π_{Mult}, $\mathcal{S}$ samples random elements as $\boldsymbol{x}+\boldsymbol{a}, \boldsymbol{y}+\boldsymbol{b}, \boldsymbol{z}+\boldsymbol{r}$, which does not influence the joint distribution of honest parties' outputs and corrupted parties' shares of $\{[z_j|j]_t\}_{j=1}^k$ due to the same property of Shamir secret sharing scheme as above. The distribution will coincide with that in real world if the verification is passed.

Note in both cases, $\mathcal{S}$ can compute corrupted parties shares of $\{[z_j|j]_t\}_{j=1}^k$ if the verifivation is passed.

- Sharing transformation (simulating Π_{ShTrans} and $\Pi_{\mathsf{VerifyShTrans}}$). If P_{king} is corrupted, when simulating Π_{ShTrans}, $\mathcal{S}$ samples random elements as honest parties' shares of $\{[\boldsymbol{x}_i + \boldsymbol{r}_i]_{n'-1}\}_{i=1}^k$, which does not change the joint distribution of honest parties' outputs and corrupted parties' shares of $\{[L_{i,j}(\boldsymbol{x}_i)|j]_t\}_{i,j=1}^k$ since honest parties' shares of $\{[\mathbf{0}_i]_{n'-1}\}_{i=1}^k$ are uniformly random. The distribution will coincide with that in the real world if the verification is passed.

  If P_{king} is honest, when simulating Π_{ShTrans}, $\mathcal{S}$ samples random elements as $\{\boldsymbol{x}_i + \boldsymbol{r}_i\}_{i=1}^k$, which does not influence the joint distribution of honest parties' outputs and corrupted parties' shares of $\{[L_{i,j}(\boldsymbol{x}_i)|j]_t\}_{i,j=1}^k$ due to the same property of Shamir secret sharing scheme again. The distribution will coincide with that in the real world if the verification is passed.

  Note in both cases, $\mathcal{S}$ can compute the corrupted parties' shares of $\{[L_{i,j}(\boldsymbol{x}_i)|j]_t\}_{i,j=1}^k$ if the verification is passed.

– In the output phase, for each group of k output gates belonging to P_s, if P_s is honest, P_s simply checks if the adversary $\mathcal{A}$ sends the corrupted parties' shares of $[\boldsymbol{z}]_{t+k-1}$ correctly. If P_s is corrupted, $\mathcal{S}$ simulates honest parties' shares of $[\boldsymbol{z}]_{t+k-1}$ by computing based on the output $\boldsymbol{z}$ received from $\mathcal{F}_{\mathsf{Main}}$ and the corrupted parties' shares of $\{[z_j|j]_t\}_{j=1}^k$. This does not change the joint distribution of honest parties' outputs and the corrupted parties' shares due to the correctness of the protocol.

□

D Enabling Preprocessing Phase

We combine the techniques proposed in [BH08] with dispute control framework to realize $\mathcal{F}_{\mathsf{RandSh}}$ and $\mathcal{F}_{\mathsf{Triples}}$ in the following.

Preparing Random Degree-t Shamir Sharings. We first introduce the definition of hyper-invertible matrix from [BH08] which will be used to prepare random linear secret sharings.

Definition 1. *An r-by-c matrix $\mathbf{M}$ is hyper-invertible if for any index sets $R \subseteq \{1, 2, \ldots, r\}$ and $C \subseteq \{1, 2, \ldots, c\}$ with $|R| = |C| > 0$, the matrix $\mathbf{M}_R^C$ is invertible, where $\mathbf{M}_R$ denotes the matrix consisting of the rows $i \in R$ of $\mathbf{M}$, $\mathbf{M}^C$ denotes the matrix consisting of the columns $j \in C$ of $\mathbf{M}$, and $\mathbf{M}_R^C = (\mathbf{M}_R)^C$.*

[GLS19] points out a very useful property of hyper-invertible matrices, which is a more generalized version compared with that shown in [BH08].

Lemma 4. *Let $\mathbf{M}$ be a hyper-invertible r-by-c matrix and $(y_1, \ldots, y_r)^{\mathrm{T}} = \mathbf{M} \cdot (x_1, \ldots, x_c)^{\mathrm{T}}$. Then for any sets of indices $A \subseteq \{1, 2, \ldots, c\}$ and $B \subseteq \{1, 2, \ldots, r\}$ such that $|A| + |B| = c$, there exists a linear function $f : \mathbb{F}^c \to \mathbb{F}^r$ which takes $\{x_i\}_{i \in A}, \{y_j\}_{j \in B}$ as inputs and outputs $\{x_i\}_{i \notin A}, \{y_j\}_{j \notin B}$.*

Applying hyper-invertible matrices, [BH08] proposed a method to verifiably prepare random linear secret sharings, which will be modified to adapt to dispute control framework. In particular, initially all parties are `happy`. First, every party P_i distributes a random linear sharing $U^{(i)}$. Then every party P_i locally multiplies

a hyper-invertible matrix to its own shares, i.e. $(U_1, \ldots, U_{n'})^{\mathrm{T}} = \mathbf{M} \cdot (U^{(1)}, \ldots, U^{(n')})^{\mathrm{T}}$, where $\mathbf{M}$ is a hyper-invertible $n' \times n'$ matrix. Then $2t'$ resulting sharings are checked, by reconstructing each sharing towards a different party who will get `unhappy` in the case of inconsistency. Then to detect the faults, after sending their happy-bits to others, all parties run a consensus protocol to agree on whether there are `unhappy` parties. If the consensus outputs `happy`, then the remaining unchecked $n' - 2t'$ sharings are outputted. Otherwise, all parties further execute a fault localization step to figure out a new disputed pair of parties containing at least one corrupted party. Then all parties redo the procedure above.

For correctness, if all honest parties get `happy`, then at least n' sharings, including t' sharings checked by honest parties and $n' - t'$ sharings shared by honest parties, are correct. Hence, all $2n'$ sharings must be correct as the remained n' sharings are linear combinations of n' correct sharings due to the property of hyper-invertible matrices stated in Lemma 4. As for its secrecy, the outputted sharings are random and unknown to the adversary because the adversary knows at most $2t'$ sharings, when fixing these $2t'$ sharings, there is a bijection between $n' - 2t'$ honest input sharings and $n - 2t'$ output shairngs.

Note the procedure above costs communication $O(n^2)$ elements to prepare random degree-t sharings of batch size $n' - 2t'$ in the case of success. Thus, to prepare N random Σ-sharings, the communication cost is $O(N \cdot n + n^4)$ elements for any $\Sigma \in \{\Sigma_{1,i}, \Sigma_{2,i,j}, \Sigma_{3,i}\}$, where $\Sigma_{3,i}$ is used to denote a kind of random sharings in the form of $([r|i]_t, [r|i]_{2t})$.

Preparing Random Beaver Triples. Before showing how to realize $\mathcal{F}_{\mathsf{Triples}}$ to prepare random packed Beaver triples, we first introduce a method also proposed in [BH08] to publicly and detectably reconstruct a batch of degree-d Shamir secret sharings with $d + t' < n'$. Actually, the protocol $\Pi_{\mathsf{VerifyPub}}$ instantiating $\mathcal{F}_{\mathsf{VerifyPub}}$ originates from this method. In particular, n' sharings $([u_1]_d, \ldots, [u_{n'}]_d)$ are expanded from a batch of T original unchecked secret sharings $([s_1]_d, \ldots, [s_T]_d)$ by multiplying a Vandermonde matrix $\mathbf{M}$ as

$$([u_1]_d, \ldots, [u_{n'}]_d)^{\mathrm{T}} = \mathbf{M} \cdot ([s_1]_d, \ldots, [s_T]_d)^{\mathrm{T}},$$

and reconstructed towards different parties who will get `unhappy` in case of inconsistency (i.e., $[u_i]_d$ is reconstructed towards P_i). Then every party P_i sends the resulting secret to all parties or sends $\perp$ if P_i is `unhappy`. For each $P_i \in \mathcal{P}$, if P_i receives at least $T + t'$ $(T-1)$-consistent values (satisfy a degree-$(T-1)$ secret sharing) in the previous step, P_i will compute $s_1, \ldots, s_T$ from any T of them and get `unhappy` otherwise. Intuitively, the correctness holds due to the super-invertibility of Vandermonde matrices.

Now all parties prepare random packed Beaver triples to realize $\mathcal{F}_{\mathsf{Triples}}(i)$ as follows.

1. All parties invoke $\mathcal{F}_{\mathsf{RandSh}}(\Sigma_{1,i})$ two times and $\mathcal{F}_{\mathsf{RandSh}}(\Sigma_{3,i})$ one time in parallel to generate three groups of sharings $\{[a_l|i]_t\}_{l=1}^T, \{[b_l|i]_t\}_{l=1}^T, \{([r_l|i]_t, [r_l|i]_{2t})\}_{l=1}^T$.
2. For $l \in [T]$, all parties locally compute $[c_l + r_l|i]_{2t} = [a_l|i]_t \cdot [b_l|i]_t + [r_l|i]_{2t}$, where $c_l = a_l \cdot b_l$.
3. All parties publicly and detectably reconstruct $\{[c_l + r_l|i]_{2t}\}_{l=1}^T$.
4. For $l \in [T]$, all parties locally compute $[c_l|i]_t = c_l + r_l - [r_l]_t$.

After finishing the procedure above, to detect the fault, all parties again sends their happy-bits to other parties and run a consensus on whether there are `unhappy` parties. If all parties are `happy`, $\{([a_l|i]_t, [b_l|i]_t, [c_l|i]_t)\}_{l=1}^T$ are outputted. Otherwise, all parties further localize a new disputed pair containing at least one corrupted party as before.

Note the security of this procedure prepraring random Beaver triples comes directly from the security of the procedure preparing random double-sharings. Note the procedure above costs communication $O(n^2)$ elements to prepare random Beaver triples of batch size $T = 2t + 1$ in the case of success. Thus, to prepare N random Beaver triples, the communication cost is $O(N \cdot n + n^4)$.

E Input Phase

We describe Π_{Input} here.

Protocol 11 : Π_{Input} in $\mathcal{F}_{\mathsf{RandSh}}, \mathcal{F}_{\mathsf{VerifyPub}}$-hybrid model

Divide all the input gates into several segments while preserving each size of the segments bounded by $|C|/n^2$. All parties handle the input gates segment by segment.
For each segment:

For every group $\boldsymbol{x} = (x_1, \ldots, x_k)$ of k input gates belonging to P_s, P_s shares its group of inputs to all parties using random sharings $\{[r_j|j]_t\}_{j=1}^k$ as follows:

1. Every party receives its shares of $\{[r_j|j]_t\}_{j=1}^k$ from $\mathcal{F}_{\mathsf{RandSh}}$.
2. Every party computes $[\boldsymbol{r}]_{t+k-1} = \sum_{j=1}^k [\boldsymbol{e_j}]_{k-1} \cdot [r_j|j]_t$ and sends it to P_s.
3. P_s checks if one of other parties sends wrong share $[\boldsymbol{r}]_{t+k-1}$: P_s only needs to check whether this is of degree $t+k-1$.
   - If it is inconsistent, every party P_i sends $\{[r_j|j]_t\}_{j=1}^k$ to P_s, $\forall i \in [n']$. P_s reconstructs $\boldsymbol{r}$, compares it with $[\boldsymbol{r}]_{t+k-1}$ to detect who is cheating and broadcasts (i, x, x'), where P_i should have sent x to P_s, but P_s claims to have received $x' \neq x$. P_s asks all parties who are responsible to transmit P_i's shares to broadcast the messages they believe. Set two adjacent parties broadcasting differently to be the dispute pair. All parties take the new dispute pair as output.
4. P_s reconstructs $\boldsymbol{r}$ using the receiving shares and distributes $[\boldsymbol{x} + \boldsymbol{r}]_{k-1}$ to all parties.
5. All parties invoke $\mathcal{F}_{\mathsf{VerifyPub}}$ to check whether P_s sends a degree-$(k-1)$ sharing.
   - If not, all parties take a new dispute pair as output.
6. For all $j \in [k]$, every party computes $[x_j|j]_t = [\boldsymbol{x} + \boldsymbol{r}]_{k-1} - [r_j|j]_t$.

F Output Phase

We describe Π_{Output} here.

Protocol 12 : Π_{Output}

Divide all the output gates into several segments while preserving each size of the segments bounded by $|C|/n^2$. All parties handle the output gates segment by segment.
For every segment,

For every group of output $\boldsymbol{z}$ belonging to P_s, every party computes $[\boldsymbol{z}]_{t+k-1} = \sum_{j=1}^k [\boldsymbol{e_j}]_{k-1} \cdot [z_j|j]_t$ and sends to P_s. P_s checks whether all parties sends a degree-$(t+k-1)$ sharing.

- If they are inconsistent, every party P_i sends its share of $\{[z_j|j]_t\}_{j=1}^k$ to P_s, $\forall i \in [n']$. P_s reconstructs $\boldsymbol{z}$, compares it with $[\boldsymbol{z}]_{t+k-1}$ to detect who is cheating and broadcasts (i, x, x'), where P_i should have sent x to P_s, but P_s claims to have received $x' \neq x$. P_s asks all parties who are responsible to transmit P_i's shares to broadcast the messages they believe. Set two adjacent parties broadcasting differently to be the dispute pair. All parties take the dispute pair as output.
- If they are consistent, P_s reconstructs its output $\boldsymbol{z}$.

G Circuit Transformation

When using packed Shamir secret sharing technique to compute a circuit, we are supposed to assume a circuit may possess several properties which are friendly for packing. For instance, in an intermediate layer, we assume the number of addition gates and the number of multiplication gates are some multiples of k so that we can group addition gates and multiplication gates into several groups of size k, respectively. More concretely, we assume a general circuit can be efficiently transformed to a circuit satisfying the following properties:

– In the input layer and the output layer, the number of input gates belonging to each party and the number of output gates belonging to each party are multiples of k. In each intermediate layer, the number of addition gates and the number of multiplication gates are multiples of k.
– 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.
– During the computation, gates that have the same type (i.e., input gates belonging to the same party, output gates belonging to the same party, 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.

We directly adopt a deterministic circuit transformation algorithm proposed in [GPS21] to efficiently obtain a resulting circuit C' which has the same output as the original circuit C and satisfies the properties above of size $|C'| = O(|C| + k \cdot (n + \mathsf{Depth}))$, where Depth is the depth of C.

As a remark, since we make use of the dispute control framework which needs to divide the circuit into several segments evaluated sequentially in order to upper bound the overhead due to re-evaluation, we are expected to further ensure the circuit of each segment is friendly to the packed Shamir sharing technique when doing segment division, i.e. the circuit of each segment should satisfy the properties listed above, as well as the requirements mentioned in Remark 1.

H Communication Complexity Analysis of Π_{Main}

Let I denote the input size, G denote the number of gates, and O denote the output size, Depth denote the depth of the circuit. Then $|C| \geq I + G + O$. We set $T = n - t, k = (n+5)/6$. We analyze the communication complexity in the online phase as follows:

– Input Phase: For every group of k inputs, the communication complexity is $O(n + n^2/T)$ elements in the case of success and $O(n^2)$ elements in the worst case where a pair of dispute parties containing at least one corrupted party must be found. Thus, the communication complexity for input phase is $O((\frac{I}{k} + n)(n + \frac{n^2}{T})) = O((\frac{I}{k} + n) \cdot n)$ elements in the case of success.
– Computation Phase:
  • Segment Randomness Preparation: For every group of k sharing transformation randomness, the communication complexity is $O(n^2k/T + nk)$ elements in the case of success and $O(n^2k)$ in the worst case where a pair of dispute parties containing at least one corrupted party must be found. For every group of $n' - t'$ degree-$(n'-1)$ $\mathbf{0}$-sharings, the communication complexity is $O(n^2)$ elements.
  • Evaluation Phase: For every group of k multiplication gates, the communication complexity is $O(n)$ elements. For every sharing transformations, the communication complexity is $O(n)$ elements.
  • Verification Phase: For every group of k multiplication gates, the communication complexity is $O(n^2/T)$ elements in the case of success and $O(n^2)$ elements in the worst case where a pair of dispute parties containing at least one corrupted party must be found. For every group of k sharing transformations, the communication complexity is $O(n^2k/T + nk)$ in the case of success and $O(n^2k)$ in the worst case where a pair of dispute parties containing at least one corrupted party must be found.

  Thus, the total communication complexity for the computation phase is $O((\mathsf{Depth} + \frac{G}{k})(n + \frac{n^2}{T} + \frac{n^2}{n'-t'})) = O((\mathsf{Depth} + \frac{G}{k}) \cdot n)$ elements in the case of success.
– Output Phase: For every group of k outputs, the communication complexity is $O(n)$ bits in the case of success and $O(nk)$ elements in the worst case where a pair of dispute parties containing at least one corrupted party must be found. Thus, the communication complexity for output phase is $O((\frac{O}{k} + n) \cdot n)$ elements in the case of success.

In summary, the overall communication complexity for online phase is $O((\mathsf{Depth} + n + \frac{|C|}{k}) \cdot n + n^2 \cdot n^2 k) = O(|C| \cdot \frac{n}{k} + (\mathsf{Depth} + n) \cdot n + n^5)$ elements, where the term n^5 is caused by redoing the segment when encountering faults.

I Instantiating Preprocessing

Instantiating Preprocessing Phase. In the preprocessing phase, all parties prepare the randomness as follows.

1. Preparing Random Masked Sharings for Input Phase: Let I denote the number of input gates. For all $i \in [k]$, let $\Sigma_{1,i}$ be the secret sharing scheme corresponding to $[r_i|i]_t$. All parties invoke $\mathcal{F}_{\mathsf{RandSh}}(\Sigma_{1,i})$ to prepare $I/k + n$ random sharings in the form of $[r_i|i]_t$.
2. Preparing Random Masked Sharings for Transpose (used for preprocessing data in sharing transformation): Let N_1 denote the number of input sharings of all intermediate layers and the output layer. For all $i, j \in [k]$, let $\Sigma_{2,i,j}$ be the secret sharing scheme corresponding to $([r_{i,j}|i]_t, [r_{i,j}|j]_t)$. All parties invoke $\mathcal{F}_{\mathsf{RandSh}}(\Sigma_{2,i,j})$ to prepare $\frac{6N_1}{k}$ random sharings in the form of $([r_{i,j}|i]_t, [r_{i,j}|j]_t)$.
3. Preparing Random Masked Sharings for Multiplication and Verifying Multiplication: Let N_2 denote the number of multiplication gates. For all $i \in [2k-1]$, all parties invoke $\mathcal{F}_{\mathsf{RandSh}}(\Sigma_{1,i})$ to prepare $\frac{2N_2}{k}$ random sharings in the form of $[r_i|i]_t$. For all $i \in [k]$, all parties invoke $\mathcal{F}_{\mathsf{RandSh}}(\Sigma_{1,i})$ to prepare $\frac{5N_2}{k}$ random sharings in the form of $[r_i|i]_t$.
4. Preparing Random Beaver Triples for Multiplication: For all $i \in [k]$, all parties invoke $\mathcal{F}_{\mathsf{Triples}}(i)$ to prepare $\frac{N_2}{k}$ Beaver triples in the form of $\{[a_i|i]_t, [b_i|i]_t, [c_i|i]_t\}$.

Analysis of the Communication Complexity of the preprocessing phase. We analyze the communication complexity of preprocessing phase as follows:

- Preparing Randomness for Input Phase: The communication to prepare $(I + n \cdot k)$ random sharings in the form of $[r_i|i]_t$ is $O((I + n \cdot k) \cdot n + n^4)$ elements.
- Preparing Randomness for Multiplication: The communication to prepare $\frac{N_2(9k-2)}{k}$ random sharings in the form of $[r_i|i]_t$ is $O(N_2 \cdot n + n^4)$ elements. The communication to prepare N_2 random Beaver triples in the form of $([a_i|i]_t, [b_i|i]_t, [c_i|i]_t)$ is $O(N_2 \cdot n + n^4)$ elements.
- Preparing Randomness for Sharing Transformation: The communication to prepare $\frac{6N_1}{k} \cdot k^2 = 6N_1 k$ random sharings in the form of $([r_{i,j}|i]_t, [r_{i,j}|j]_t)$ is $O(N_1 nk + n^4)$ elements. Then the communication for sharing transformations is $O(N_1 nk + n^4)$ elements.

Since $N_1 \leq G/k + \mathsf{Depth}, N_2 \leq G + \mathsf{Depth} \cdot k$, then the communication for preprocessing phase is $O((I + n \cdot k + N_2 + N_1 \cdot k)n + n^4) = O((I + n \cdot k + k \cdot \mathsf{Depth} + G)n + n^4) = O((|C| + k(n + \mathsf{Depth}))n + n^4)$ elements.

J Segment Division

We formally describe the procedure for segment division after finishing the circuit transformation in Algorithm 1 below.

Algorithm 1: Segment Division

Input: A circuit C of circuit depth Depth with $|C|/k$ groups of gates, $g_0, \ldots, g_{|C|/k-1}$ labelled following the circuit topology. For a group g of k gates, denote the set of wires copied from the output wires of g by g_{out}.

Output: $S, \mathsf{Segment}[0 : \mathsf{S} - 1]$

Initialize $i \leftarrow 0, \mathsf{Segment}[i] \leftarrow \emptyset, \mathsf{SegSize} = 0, \mathsf{SegDepth} = 0$;

for $j = 0$ *to* $|C|/k - 1$ **do**

  // Check the requirements of the segment and keep adding groups of gates until the segment is full or all gates are included.

  if $\mathsf{SegSize} + k \leq |C|/n^2$ *and* $\mathsf{SegDepth} + \mathbb{1}_{g_j \textit{ belongs to a new layer}} \leq \mathsf{Depth}/n^2$ **then**

    $\mathsf{Segment}[i] \leftarrow \mathsf{Segment}[i] \cup \{g_j\}$;

    $\mathsf{SegSize} \leftarrow \mathsf{SegSize} + k$;

    $\mathsf{SegDepth} \leftarrow \mathsf{SegDepth} + \mathbb{1}_{g_j \text{ belongs to a new layer}}$;

    if $j \neq |C|/k - 1$ **then**

      Continue ;

  // After determining the segment, handle fan-out gates of the segment.

  $\mathsf{fanout\text{-}rest} \leftarrow \emptyset$;

  for *each* g *in* $\mathsf{Segment}[i]$ **do**

    Consider all wires in g_{out} ;

    // Handle the fan-out gates that will be used in the current segment.

    Collect a set F of all the wires that will be used as input wires in $\mathsf{Segment}[i]$, add the first $k \cdot \lceil \frac{|F|}{k} \rceil - |F|$ wires in $g_{out} \setminus F$ to F in order to make the size of F a multiple of k, and assign each k wires in F a fan-out gate which will be included in $\mathsf{Segment}[i]$;

    For the rest wires in g_{out}, assign each k of them a fan-out gate which will be temporarily included in $\mathsf{fanout\text{-}rest}$;

  // Handle the rest of the fan-out gates.

  For all fan-out gates in $\mathsf{fanout\text{-}rest}$, uniformly divide them into $m = \lceil |\mathsf{fanout\text{-}rest}| / \lfloor \frac{|C|}{n^2 \cdot k} \rfloor \rceil$ segments, $\mathsf{segment}[i+1], \ldots, \mathsf{segment}[i+m]$, with each segment containing at most $\lfloor \frac{|C|}{n^2 \cdot k} \rfloor$ fan-out gates ;

  // Start the next segment division.

  $i \leftarrow i + m + 1, \mathsf{Segment}[i] \leftarrow \emptyset, \mathsf{SegSize} = 0, \mathsf{SegDepth} = 0$;

$S = i$;

Output $S, \mathsf{Segment}[0 : S - 1]$;