Communication-Efficient Unconditional MPC with Guaranteed Output Delivery

Vipul Goyal[1], Yanyi Liu[2], and Yifan Song[1](✉)

[1] Carnegie Mellon University, Pittsburgh, USA
vipul@cmu.edu, yifans2@andrew.cmu.edu
[2] Tsinghua University, Beijing, China
yl2866@cornell.edu

Abstract. We study the communication complexity of unconditionally secure MPC with guaranteed output delivery over point-to-point channels for corruption threshold $t < n/3$. We ask the question: "is it possible to construct MPC in this setting s.t. the communication complexity per multiplication gate is linear in the number of parties?" While a number of works have focused on reducing the communication complexity in this setting, the answer to the above question has remained elusive for over a decade.

We resolve the above question in the affirmative by providing an MPC with communication complexity $O(Cn\kappa + n^3\kappa)$ where κ is the size of an element in the field, C is the size of the (arithmetic) circuit, and, n is the number of parties. This represents a strict improvement over the previously best known communication complexity of $O(Cn\kappa + D_M n^2\kappa + n^3\kappa)$ where D_M is the multiplicative depth of the circuit. To obtain this result, we introduce a novel technique called *4-consistent tuples of sharings* which we believe to be of independent interest.

1 Introduction

In secure multiparty computation (MPC), a set of n players wish to evaluate a function f on their private inputs. The function f is publicly known to all players and is assumed to be an arithmetic circuit C over some finite field. Very informally, the protocol execution should not leak anything about the individual inputs beyond what can already be inferred from the function output.

The notion of MPC was introduced in the beautiful work of 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 [BOGW88,CCD88] considered the unconditional (or information-theoretic) setting and showed a positive result for up to $t < n/3$ corrupted parties assuming point-to-point communication channels. If one assumes a broadcast channel in addition, it was

V. Goyal, Y. Song—Research supported in part by a JP Morgan Faculty Fellowship, a gift from Ripple, a gift from DoS Networks, a grant from Northrop Grumman, and, a Cylab seed funding award.

shown in [RBO89,Bea89] how to obtain positive results in the unconditional setting for up to $t < n/2$ corrupted parties. MPC plays a central role in cryptography, and, by now has been studied in a variety of different interesting settings. Examples include security against semi-honest adversaries vs malicious adversaries, unconditional security vs computational security, and, security with abort vs guaranteed output delivery.

In this work, we are interested in the communication complexity of unconditionally secure MPC with guaranteed output delivery. We assume that the parties are connected using point-to-point channels, the adversary is malicious (and may deviate from the protocol arbitrarily), and, the corruption threshold $t < n/3$ (to avoid the known negative results on Byzantine agreement [LSP82]). Indeed, the classical BGW protocol already gives a feasibility result in this setting by presenting an unconditional MPC with guaranteed output delivery for corruption threshold $t < n/3$. Several subsequent works have focused on improving the communication complexity of MPC in this setting. Note that the real world efficiency of MPC in the unconditional setting is typically dominated by its communication complexity (as opposed to the computational complexity). This is because the local computations required are typically simple: often just a series of linear operations. Representing the functionality as an arithmetic circuit, the addition gates are typically "free" requiring no communication at all. Hence, the communication complexity of the protocol depends upon the number of multiplication gates in the circuit.

In this paper, we ask the following natural question:

"Is it possible to construct unconditional MPC with guaranteed output delivery for $t < n/3$ s.t. the communication complexity per multiplication gate is linear in the number of parties?"

Having linear communication complexity is interesting as it means that the work done by a single party is independent of the number of parties participating in the computation, giving a fully scalable protocol. While a number of works have made significant progress, the answer to this question has remained elusive so far. Best known communication complexity for malicious adversaries in this setting comes from the construction in [BTH08]. The construction in [BTH08] has communication complexity $O(Cn\kappa + D_M n^2\kappa + n^3\kappa)$ where κ is the size of an element in the field, C is the circuit size and D_M is the multiplicative depth of the circuit. For circuits which are "narrow and deep" (i.e., the multiplicative depth D_M is not much smaller than the circuit size C), the communication complexity per multiplication gate can be as high as $O(n^2)$ elements. The factor of $D_M n^2\kappa$ unfortunately appears in several papers studying this setting [DN07,BTH08], [DIK10,BSFO12]. This led Ben-Sasson, Fehr, and, Ostrovsky to ask the question whether this factor is inherent [BSFO12].

Building on a variety of previous works, it was shown in [DI06,HN06] that there exist cryptographic secure MPC protocols with linear communication complexity with guaranteed output delivery. In the unconditional setting, linear communication complexity protocols are known for passive adversaries [DN07].

However the question of obtaining an analogous construction against active adversaries has remained open for over a decade.

Our Results. In this work, we resolve the above question in the affirmative by providing an MPC with guaranteed output delivery, perfect security against a malicious adversary corrupting up to $t < n/3$ of the parties, and, using only point-to-point communication channels. The communication complexity of our construction is $O(Cn\kappa + n^3\kappa)$ which is a strict improvement over the previous best result of [BTH08]. Compared with the work [BTH08], our main contribution is removing the quadratic term related to the multiplicative depth of the circuit, while keeping the circuit-independent term as efficient as that in [BTH08]. To obtain this result, we introduce a novel technique which we call *4-consistent tuples of sharings*. Very informally, this technique allows us to increase the redundancy in an n-out-of-n sharing "on demand" such that if an adversary cheats and changes its share in the n-out-of-n sharing, it can be detected and either kicked out or added to a list of disputed parties. A high level overview of the key technical obstacle encountered by previous works, and, how we overcome it using 4-consistent tuples of sharings is given in Section 2.

Related Works. The notion of MPC was first introduced in [Yao82,GMW87] in 1980s. Feasibility results for MPC were obtained by [Yao82,GMW87,CDVdG87] under cryptographic assumptions, and by [BOGW88,CCD88] in the information-theoretic setting. Subsequently, a large number of works have focused on improving the efficiency of MPC protocols in various settings.

In this work, we focus on improving the asymptotic communication complexity of MPC for arithmetic circuits over a finite field with output delivery guarantee and security against an active adversary which may control up to $t < n/3$ parties, in the information-theoretical setting. After the pioneering work of Ben-Or et al. [BOGW88] which shows the feasibility in this setting, Hirt et al. [HMP00] introduced Party-Elimination Framework which is a general technique to efficiently transform a semi-honest protocol into a protocol providing unconditional security with minimal additional cost. With this technique, Hirt et al. constructed a MPC protocol with communication complexity $O(Cn^3\kappa + \text{poly}(n, \kappa))$ bits, where C is the size of the circuit and κ is the size of an element in the underlying field. A number of works [HM01,DN07,BTH08] then continued to improve the communication complexity by using this technique. The previous best result [BTH08] provided a protocol with perfect security with asymptotic communication complexity $O(Cn\kappa + D_M n^2\kappa + n^3\kappa)$ bits, where D_M is the multiplicative depth of the circuit. In a subsequent result [BSFO12] which focuses on the setting of security against up to $t < n/2$ corrupted parties assuming the existence of a broadcast channel, the authors raised the question whether the quadratic dependency w.r.t. the multiplicative depth is an inherent restriction. Our result answers this question by presenting the first construction which achieves linear communication per multiplication gate.

A number of works also focus on improving the communication efficiency of MPC with output delivery guarantee in the settings with different threshold on

the number of corrupted parties. In the setting where $t < (1/3 - \epsilon)n$, secret sharing can be used to hide a batch of values, resulting in more efficient protocols. E.g., Damgard et al. [DIK10] introduced a protocol with communication complexity $O(C \log C \log n \cdot \kappa + D_M^2 \text{poly}(n, \log C)\kappa)$ bits. In the setting of honest majority (i.e., $t < 1/2n$), Hirt et al. [HN06] presented a protocol with communication complexity $O(Cn\kappa + n\mathcal{BC})$, where $\mathcal{BC}$ is the cost for broadcasting a bit by one party, by using threshold homomorphic encryption [Pai99] and assuming the existence of a broadcast channel. Ben-Sasson et al. [BSFO12] presented a protocol with communication complexity $O(Cn\kappa + D_M n^2\kappa + n^7\kappa)$ assuming the existence of a broadcast channel in the information-theoretical setting. More recent works in the computational setting have been able to obtain communication efficient MPC with output delivery guarantee in as low as 3 rounds, e.g., [BJMS18].

A rich line of works have focused on the performance of MPC in practice. Many concretely efficient MPC protocols were presented in [LP12,NNOB12,FLNW17] [ABF+17,LN17,CGH+18]. All of these works emphasized on the practical running time and only provided security with abort. Some of them were specially constructed for two parties [LP12,NNOB12] or three parties [FLNW17,ABF+17].

2 Technical Overview

Our goal is to construct an unconditionally secure MPC protocol with guaranteed output delivery against a fully malicious adversary which may corrupt $t < n/3$ parties. Our construction is for arithmetic circuits over a finite field, and, achieves a communication complexity per multiplication gate which is linear in the number of parties. The previous best result in this setting was obtained over a decade ago by [BTH08]. In this section, we present an overview of our main ideas.

How Previous Techniques Work: The communication complexity achieved by the construction in [BTH08] is $O(Cn\kappa + D_M n^2\kappa + n^3\kappa)$ where κ is the size of an element in the field, C is the circuit size and D_M is the multiplicative depth of the circuit. Our goal would be to eliminate the term $O(D_M n^2\kappa)$ which would allow us to obtain a construction with linear communication complexity per multiplication gate.

We now take a closer look at the construction in [BTH08]. To improve the efficiency, several subroutines in [BTH08] handle a *batch* of $O(n)$ multiplication gates at one time. The overall cost of each such operation is $O(n^2)$ elements. In this way, the amortized cost per multiplication gate only comes out to $O(n)$ elements. While this seems to already give us the result we seek, a major limitation of the techniques in [BTH08] (as well as prior works) is that the batches must solely consist of multiplication gates *at the same depth in the circuit*. Since the communication cost for a batch is at least $O(n^2)$ field elements (even if the number of multiplication gates in the batch is significantly lower than n),

and, the number of batches is at least the multiplicative depth D_M, the overall communication complexity cannot be lower than $O(D_M n^2 \kappa)$.

The fundamental reason why we can only handle multiplication gates from a single layer in any given batch is that the parties need to ensure that the result of computing a layer is correct before moving on to the next layer. To understand why this is the case, consider the following explicit attack.

The Key Bottleneck: We briefly describe the protocol for each multiplication gate in [BTH08]. Let x, y be the inputs of the multiplication gate. We use $[x]_d$ to denote a d-sharing of x. A d-sharing of x is the vector of shares obtained by applying a $(d+1)$-out-of-n secret sharing scheme on x. After a sharing is distributed, each party holds a single share. Now consider the evaluation of a multiplication gate. In the beginning, all parties hold shares of input wire values $[x]_t, [y]_t$. In addition, all parties also hold a random (Beaver) tuple of sharings $([a]_t, [b]_t, [c]_t)$ where $c = ab$ generated in the preparation phase. To compute the output sharing $[xy]_t$, the parties will go through the following steps:

1. All parties locally compute $[x+a]_t := [x]_t + [a]_t$ and $[y+b]_t := [y]_t + [b]_t$.
2. All parties reconstruct $[x+a]_t, [y+b]_t$ by using a reconstruction protocol discussed below.
3. On receiving $x+a, y+b$, all parties locally compute $[xy]_t = (x+a)(y+b) - (x+a)[b]_t - (y+b)[a]_t + [c]_t$.

Our first attempt is to let one party $P_{\texttt{king}}$ collect all the shares from $[x+a]_t, [y+b]_t$, reconstruct $x+a, y+b$, and send the results back to all other parties. In this way, each multiplication gate only costs $O(n\kappa)$ bits even though we are evaluating a single multiplication gate. Even if some of the parties are corrupted, $P_{\texttt{king}}$ can use error correction to recover the correct values. However, if $P_{\texttt{king}}$ itself is corrupted, the honest parties may get incorrect results. In fact, $P_{\texttt{king}}$ may even decide to send different values to different honest parties resulting in honest parties holding inconsistent shares. At this point, if without any further verification, one more multiplication gate is computed on the resulting output, we show that $P_{\texttt{king}}$ can learn the (full) value on an internal wire of the circuit!

In more detail, suppose $P_{\texttt{king}}$ is corrupted. For a sharing $[a]_t$, we use a_i to denote the i-th share of $[a]_t$. All parties are going to evaluate $x \cdot y$ and then $(xy) \cdot z$. We give an attack to allow the adversary to recover the value of y:

1. $P_{\texttt{king}}$ receives all shares of $[x+a]_t$ and $[y+b]_t$ from all parties. $P_{\texttt{king}}$ computes $(x+a)$ and $(y+b)$.
2. $P_{\texttt{king}}$ selects a set of honest parties $\mathcal{H}'$ with size $|\mathcal{H}'| = n - t - 1$.
3. $P_{\texttt{king}}$ sends $(x+a)$ and $(y+b)$ to parties which are not in $\mathcal{H}'$. For parties $P_j \in \mathcal{H}'$, $P_{\texttt{king}}$ sends $(x+a+1)$ and $(y+b)$.
4. All parties locally compute $[xy]_t = (x+a)(y+b) - (x+a)[b]_t - (y+b)[a]_t + [c]_t$. For a party $P_j \in \mathcal{H}'$, the share of $[xy]_t$ it should hold is $(x+a)(y+b) - (x+a)b_j - (y+b)a_j + c_j$. However, since $P_{\texttt{king}}$ sent $(x+a+1)$ instead of $(x+a)$ to P_j, the actual share P_j holds is $((x+a)(y+b) - (x+a)b_j - (y+b)a_j + c_j) + (y+b-b_j)$. This is equal to the correct share of $[xy]_t$ plus the value $(y+b-b_j)$. A party P_k which is not in $\mathcal{H}'$ holds the correct share of $[xy]_t$.

5. For the next multiplication gate, $P_{\texttt{king}}$ receives shares of $[xy+a']_t$ and $[z+b']_t$ from all the parties. For $[xy+a']_t$, $P_{\texttt{king}}$ uses the $(t+1)$ shares provided by parties (including those controlled by the adversary) that are not in $\mathcal{H}'$ to reconstruct $(xy+a')$. Then $P_{\texttt{king}}$ computes the correct shares parties in $\mathcal{H}'$ should hold.
6. For each party $P_j \in \mathcal{H}'$, $P_{\texttt{king}}$ computes the difference between the correct share of P_j and the real share P_j provided, which is $(y+b-b_j)$. Note that $P_{\texttt{king}}$ learnt $y+b$ while evaluting the previous multiplication gate, and therefore, $P_{\texttt{king}}$ learns the value of b_j.
7. The adversary uses the shares of $[b]_t$ held by corrupted parties and $\{b_j\}_{P_j \in \mathcal{H}'}$, the shares of parties in $\mathcal{H}'$, to reconstruct the value of b. Then it can compute y from $y+b$ and b. If y was the value on an input wire, the adversary has learnt an input value. Else it learnt an intermediate wire value in the circuit.

Using n-out-of-n Secret Sharing: The above attack works because of the inherent redundancy in a t-sharing. By only learning a small number of shares, the adversary can compute the correct values of the remaining shares, obtain the (incorrect) values for these shares, and finally, learn private information by comparing the incorrect values to the correct values. Our natural starting point to fix this problem would be to use a $(n-1)$-sharing (i.e., n-out-of-n sharing). In the preparation phase, we generate a random tuple of sharings $([a]_{t,n-1}, [b]_{t,n-1}, [c]_t)$ where $[a]_{t,n-1}$ denotes a t-sharing, and, an $(n-1)$-sharing of the same value a. The parties locally compute $[x+a]_{n-1} = [x]_t + [a]_{n-1}$ (instead of $[x+a]_t$). Now it can be shown that each share an honest party sends to $P_{\texttt{king}}$ is uniformly distributed, and, the above attack ceases to work. The parties can safely evaluate *multiple layers of multiplication gates* without leaking any information to the adversary.

While this idea fixes the attack we outlined earlier, eliminating the redundancy unfortunately opens the door to a host of other attacks which we discuss next.

Checking the Reconstructions: An obvious issue with the above approach is now not only $P_{\texttt{king}}$, but *any* party can cheat by sending a wrong share to $P_{\texttt{king}}$. As before, a corrupted $P_{\texttt{king}}$ can also send incorrect values and even different values to different parties. Therefore, we need to run a verification procedure to ensure every party behaved honestly. However before running a verification, all parties evaluate exactly $O(n)$ multiplication gates (even though they may not be at the same layer). The verification is thus done in batches.

First, the parties check whether (for each multiplication gate) they all received the same elements from $P_{\texttt{king}}$. This verification is done in batches and is based on techniques from prior works. If this check fails, a pair of disputed parties is identified and removed, and, all multiplication gates are re-evaluated. Otherwise, this check guarantees that the sharings of the output of all the multiplication gates held by honest parties are consistent (though not necessarily correct).

Next, the parties check whether the reconstructions of values $[x+a]_{n-1}, [y+b]_{n-1}$ are correct. Towards that end, we will use the reconstruction protocol from [BTH08] to reconstruct $[x+a]_t := [x]_t + [a]_t, [y+b]_t := [y]_t + [b]_t$, which guarantees that all honest parties get the correct results. If this second check passes, then all multiplication gates are correctly evaluated and all parties continue to evaluate the remaining multiplication gates.

However if the above check fails, we run into an obstacle. In [BTH08], failure of this check would necessarily imply the dishonesty of $P_{\texttt{king}}$ (since the messages sent by other parties to $P_{\texttt{king}}$, even if maliciously generated, could be corrected by relying on the redundancy). Thus, $P_{\texttt{king}}$ would be kicked out and the whole batch would be executed again. However in our setting, there are two possibilities: 1) at least one corrupted parties sent a wrong share to $P_{\texttt{king}}$, or, 2) $P_{\texttt{king}}$ distributed wrong results. Even if one is given that $P_{\texttt{king}}$ behaved honestly, it is hard for anybody (including $P_{\texttt{king}}$) to tell which party sent the wrong share. This is because the shares do not have any redundancy and it is possible to change the secret without getting detected by just changing a single share.

Without loss of generality, let $(x^{i^\star} + a^{i^\star})$ be the first wrong value reconstructed by $P_{\texttt{king}}$ (the parties learn $i^\star$ as part of the above check).

Increasing Redundancy using 4-Consistent Tuples: Observe that if a party supplied an incorrect share of $[x^{i^\star} + a^{i^\star}]_{n-1}$, then since $(x^{i^\star} + a^{i^\star})$ was shared using a $(n-1)$-sharing, the only way to detect who is cheating would be to go back to how these shares were generated, recompute the correct share for each party, and, see which party supplied an incorrect share. Note that $[x^{i^\star} + a^{i^\star}]_{n-1} = [x^{i^\star}]_t + [a^{i^\star}]_{n-1}$ and $[x^{i^\star}]_t$ is a t-sharing. Therefore, we focus on the generation process of $[a^{i^\star}]_{n-1}$.

The generation of $[a^i]_{n-1}$ is done in batches as follows [BTH08]. Each party P_i first randomly generates $[s^i]_{t,n-1}$ for a random element s^i and distributes the sharings to all other parties (i.e., j-th share to P_j). Then all parties extract the randomness by using a hyper-invertible matrix M and locally computing $([a^1]_{t,n-1}, \ldots, [a^n]_{t,n-1}) = M([s^1]_{t,n-1}, \ldots, [s^n]_{t,n-1})$. In particular, $[a^{i^\star}]_{n-1} = M_{i^\star}([s^1]_{n-1}, \ldots, [s^n]_{n-1})$, where $M_{i^\star}$ is the $i^\star$-th row of M.

Our first attempt to resolve this problem is as follows. The sharing $[s^i]_t$ can be seen as the "redundant" version of the sharing $[s^i]_{n-1}$. Similarly, the matrix $([s^1]_t, \ldots, [s^n]_t)$ can be seen as the redundant version of $([s^1]_{n-1}, \ldots, [s^n]_{n-1})$. The parties can generate the redundant version of $[a^{i^\star}]_{n-1}$ as $[a^{i^\star}]_t = M_{i^\star}([s^1]_t, \ldots, [s^n]_t)$. The parties can now send the shares from $[x^{i^\star} + a^{i^\star}]_t$ to $P_{\texttt{king}}$. These shares cannot be modified by the adversary because of the large redundancy present. However what if a party cheated while sending shares from $[x^{i^\star} + a^{i^\star}]_{n-1}$ but not while sending shares from $[x^{i^\star} + a^{i^\star}]_t$? The goal of detecting who cheated still continues to evade us.

To resolve this problem, we wish to create a redundant version of the matrix $([s^1]_{n-1}, \ldots, [s^n]_{n-1})$ in a way such that from this version, the *entire* matrix can be recovered. That is, for each i, we should be able to recover the entire sharing $[s^i]_{n-1}$ as opposed to just the secret s^i (even if the adversary tampers with the shares it holds). Towards that end, our idea would be to actually convert this

given matrix into *three* separate matrices such that each row of these matrices is a carefully chosen t-sharing. Even if the adversary tampers with its shares arbitrarily in each of these 3 matrices and the original matrix, these 3 matrices can be entirely recovered and then, be used to recover the original matrix. We now give more details.

All parties first agree on a partition of the set of all parties $\mathcal{P} = \mathcal{P}_1 \bigcup \mathcal{P}_2 \bigcup \mathcal{P}_3$ such that $|\mathcal{P}_1|, |\mathcal{P}_2|, |\mathcal{P}_3| \leq t + 1$. For the first matrix, the columns held by parties in $\mathcal{P}_1$ are the same as the original matrix and the remaining columns are randomly sampled such that each row of the matrix is a t-sharing. It can always be achieved since for each row, only up to $t+1$ values are fixed (i.e., copied from the original matrix). Similarly, for the second and the third matrices, the columns held by parties in $\mathcal{P}_2$ and $\mathcal{P}_3$ are the same as the original matrix respectively. The remaining columns are randomly sampled such that each row of these two matrices is a t-sharing. To recover the original matrix from these 3 matrices, we simply pick the columns held by parties in $\mathcal{P}_1$ from the first matrix, the columns held by parties in $\mathcal{P}_2$ from the second matrix and the columns held by parties in $\mathcal{P}_3$ from the third matrix. In case the adversary tampers with up to t columns of each matrix, all the 3 matrices can be recovered using error correction and then the original matrix can be recovered.

We now focus on the i-th row of these four matrices (including the original one). Denote these by $([_0s^i]_{n-1}, [_1s^i]_t, [_2s^i]_t, [_3s^i]_t)$ (recall that each row of each matrix is a sharing). Together with the t-sharing $[_0s^i]_t := [s^i]_t$, we call such 4 sharings $([_0s^i]_t, [_1s^i]_t, [_2s^i]_t, [_3s^i]_t)$ a 4-consistent tuple of sharings. More formally, we say a tuple of sharings $([_0s]_t, [_1s]_t, [_2s]_t, [_3s]_t)$ is 4-consistent w.r.t. a partition of $\mathcal{P} = \mathcal{P}_1 \bigcup \mathcal{P}_2 \bigcup \mathcal{P}_3$ where $|\mathcal{P}_1|, |\mathcal{P}_2|, |\mathcal{P}_3| \leq t + 1$, if the $(n-1)$-sharing $[s]_{n-1}$, where the k-th share of $[s]_{n-1}$ equals the k-th share of $[_js]_t$ for all $j \in \{1, 2, 3\}$ and $P_k \in \mathcal{P}_j$, satisfies that $s = {}_0s$.

We prove that 4-consistency is preserved under linear operations. In more detail, by applying $M_{i^\star}$ to the 3 new matrices, we are able to obtain three t-sharings $[_1a^{i^\star}]_t, [_2a^{i^\star}]_t, [_3a^{i^\star}]_t$ such that these would entirely allow one to recover all shares of $[a^{i^\star}]_{n-1}$ and make sure $[a^{i^\star}]_{n-1}$ and $[a^{i^\star}]_t$ are sharings of the same value $a^{i^\star}$. We stress that these 3 sharings not only allow us to recover all shares of $[a^{i^\star}]_{n-1}$, but, in fact, also provide sufficient redundancy to make sure that an adversary controlling up to t parties cannot cause the recovery procedure to fail.

Using 4-Consistent Tuples to Detect the Cheaters: How do the parties generate these 4 matrices, and, ensure 4-consistency? Each party P_i generates the sharings $[s^i]_{t,n-1}$ and distributes them as before. P_i generates three additional t-sharings $[_1s^i]_t, [_2s^i]_t, [_3s^i]_t$ such that for $j \in \{1, 2, 3\}$ and $P_k \in \mathcal{P}_j$, the k-th share of $[_js^i]_t$ equals the k-th share of $[s^i]_{n-1}$. P_i then distributes $[_js^i]_t$ to all parties which are not in $\mathcal{P}_j$ (because parties in $\mathcal{P}_j$ have already received their shares when P_i distributed $[s^i]_{n-1}$) for every $j \in \{1, 2, 3\}$. Let $[_0s^i]_t := [s^i]_t$.

Next, all parties must check whether each party P_i distributed a valid 4-consistent tuple of sharings $([_0s^i]_t, [_1s^i]_t, [_2s^i]_t, [_3s^i]_t)$. We develop subroutines to do it efficiently by checking a batch of them each time. (If the check fails, a pair of disputed parties is identified and removed.)

Recall that $[a^{i^\star}]_{n-1} = M_{i^\star}([s^1]_{n-1}, \ldots, [s^n]_{n-1})$. To verify whether parties in $\mathcal{P}_j$ provided correct shares when reconstructing $x^{i^\star} + a^{i^\star}$, all parties (locally) compute

$$\begin{aligned}[_jd^{i^\star}]_t &:= [x^{i^\star}]_t + [_ja^{i^\star}]_t \\ &= [x^{i^\star}]_t + M_{i^\star}([_js^1]_t, \ldots, [_js^n]_t)\end{aligned}$$

The computed shares are then sent to $P_{\texttt{king}}$. Observe that for $P_k \in \mathcal{P}_j$, the k-th shares of $[_js^1]_t, \ldots, [_js^n]_t$ are exactly the k-th shares of $[s^1]_{n-1}, \ldots, [s^n]_{n-1}$. On receiving all shares of $[_jd^{i^\star}]_t$, $P_{\texttt{king}}$ is able to recover all shares of $[_jd^{i^\star}]_t$ even if several of the received shares are incorrect. This allows $P_{\texttt{king}}$ to recover correct shares from $[x^{i^\star}]_t + [_ja^{i^\star}]_{n-1}$ for all $P_k \in \mathcal{P}_j$. $P_{\texttt{king}}$ can now check whether a party in $\mathcal{P}_j$ behaved honestly by sending the correct share earlier. Therefore, in the end, $P_{\texttt{king}}$ claims that some party P_k is corrupted and all parties treat $(P_{\texttt{king}}, P_k)$ as a pair of disputed parties.

On the Proof of Security: We point out that it is non-trivial to prove the security of our construction. Recall that each party P_i generates a tuple of t-sharings to encode its randomness when generating $[a^{i^\star}]$ (i.e., to encode $[s^i]_{n-1}$). In general, a t-sharing can only be used to hide one value since the adversary might have t shares and just need one more to reconstruct all shares and the secret value. However, we use a t-sharing to encode up to $t+1$ values, and, the values held by honest parties should remain unknown to the adversary. Therefore, one must be careful in using 4-consistent tuples to ensure that the simulator is able to obtain an identical view. For more details, we refer the readers to Appendix A.

Efficiency: In each batch, $O(n)$ multiplication gates are first evaluated. Then all parties check whether the results are correct. When a pair of disputed parties is identified, these two parties are removed and all these $O(n)$ multiplication gates will be reevaluated. Our protocol costs $O(n^2\kappa)$ bits for the entire batch. Therefore, on average, each multiplication only costs $O(n\kappa)$ bits. For each failure, at least one corrupted party is removed. Thus the number of reevaluations is bounded by $O(n)$, which means that reevaluations cost at most $O(n^3\kappa)$. Hence, the communication complexity of the overall protocol is $O(Cn\kappa + n^3\kappa)$.

3 Preliminary

3.1 Model

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

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, random, and output gates. Let $\kappa = \log |\mathbb{F}|$ be the size of an element in $\mathbb{F}$.

An adversary is able to corrupt at most $t < n/3$ 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 $\mathcal{C}$.

Each party P_i is assigned with a unique non-zero field element $\alpha_i \in \mathbb{F}\backslash\{0\}$ as the identity.

3.2 Byzantine Agreement

Byzantine agreement allows all honest parties to reach a binary consensus. A protocol for byzantine agreement takes a bit from each party as input, and all honest parties will reach to a consensus if at most t parties are corrupted. Furthermore, if all honest parties hold the same bit b in the beginning, then all honest parties agree on b finally.

In our protocol, we use a byzantine agreement protocol to let all parties reach a binary consensus and let one party broadcast one bit to all other parties. Broadcast allows a party (as a sender) to send a bit b to the remaining parties and all parties eventually receive the same bit b' where $b = b'$ when the sender is honest. An easy way to instantiate broadcast is to let the party send the bit b to all other parties, and then all parties run a byzantine agreement protocol to reach a consensus on the bit b' they received.

With $t < n/3$, both consensus and broadcast can be achieved by a perfect byzantine agreement protocol communicating $O(n^2)$ bits [BGP92,CW92].

3.3 Party-Elimination Framework

Party-Elimination was first introduced in [HMP00]. It is a general strategy to achieve perfect security efficiently.

The basic idea is to divide the computations into several segments. For each segment, all active parties first evaluate this segment and then check the correctness of the evaluation. It is guaranteed that at least one honest party will discover that the segment is evaluated incorrectly if any corrupted parties deviate from the protocol. After the check is completed, all active parties reach a consensus on whether this segment is successfully evaluated. In the case of success, all active parties continue to evaluate the next segment. In the case of failure, all active parties run another protocol to locate two active parties such that at least one of them is corrupted. Then these two parties are eliminated from the set of active parties. The same segment is evaluated again.

Therefore, each failure results in a reduction in the number of corrupted parties and only a bounded number ($O(n)$) of failures may happen.

We use $\mathcal{P}_{\texttt{active}}$ to denote the set of parties which are active currently. Only parties in $\mathcal{P}_{\texttt{active}}$ can participate in the remaining computations. We use $\mathcal{C}_{\texttt{active}} \subset \mathcal{P}_{\texttt{active}}$ for the set of active corrupted parties. Let n' be the size of $\mathcal{P}_{\texttt{active}}$. We use t' for the maximum possible number of the corrupted parties in $\mathcal{P}_{\texttt{active}}$.

Each time a pair of disputed parties is identified, these two parties are removed from $\mathcal{P}_{\texttt{active}}$ and hence $\mathcal{C}_{\texttt{active}}$. It results in $n' := n' - 2$ and $t' := t' - 1$.

Initially we have $n = n', t = t'$. Let $T = n' - 2t'$. Therefore, T remains unchanged during the whole protocol.

We directly borrow the instantiation of Party-Elimination Framework used in [BTH08]. We build a compiler which takes a procedure π as input and outputs a procedure or a protocol π' which either outputs the original result of π or outputs a pair of disputed parties which contains at least one corrupted party. In the rest of the constructions, each party maintains a happy-bit. Formally,

Procedure 1 PARTY-ELIMINATION(π)

1: Initialization Phase:
   All parties initially set their happy-bits to `happy`.
2: Computation Phase:
   All parties run the procedure π.
3: Fault Detection Phase:
   1. Each party sends its happy-bit to all other parties.
   2. For each party, if at least one of the happy-bits it receives is `unhappy`, sets its happy-bit to be `unhappy`.
   3. All parties run a consensus protocol on their respective happy-bits. If the result is `happy`, all parties take the result of π as the output and halt. Otherwise, run the following steps.
4: Fault Localization Phase:
   1. All parties agree on a referee $P_r \in \mathcal{P}_{\texttt{active}}$. Every other party sends everything it generated, sent, and received in the Computation Phase and Fault Detection Phase to P_r.
   2. On receiving all information from other parties, P_r simulates the Computation Phase and Fault Detection Phase. P_r broadcasts either $(P_i, \text{corrupt})$ (in the case P_i does not follow the procedure) or $(\ell, P_i, P_k, v, v', \text{disputed})$ where ℓ is the index of the message where P_i should have sent v to P_k while P_k claimed to have received $v' \neq v$.
      (a) If $(P_i, \text{corrupt})$ is broadcast, all parties set $E = \{P_r, P_i\}$.
      (b) Otherwise, P_i and P_k broadcast whether they agree with P_r. If P_i disagrees, set $E = \{P_r, P_i\}$; if P_k disagrees, set $E = \{P_r, P_k\}$; otherwise, set $E = \{P_i, P_k\}$.
   3. All parties take E as the output and halt.

We point out that the happy-bits are used in π and therefore, the value of a happy-bit reflects whether this party is satisfied with the execution of π.

After a procedure π is compiled by PARTY-ELIMINATION, parties will communicate with each other in Fault Detection Phase and Fault Localization Phase, which adds some communication cost to π'. Parties will communicate $O(n^2)$ elements in Fault Detection Phase to distribute happy-bits and reach the consensus. Let $m(\pi)$ be the total elements communicated in π, the overhead of the Fault Localization Phase will then be $O(m(\pi) + n^2)$. In total, the overall communication complexity is $O(m(\pi) + n^2)$ elements or $O(m(\pi)\kappa + n^2\kappa)$ bits.

3.4 Hyper-Invertible Matrix

We adopt the definition of hyper-invertible matrices from [BTH08].

Definition 1 ([BTH08]). *An r-by-c matrix M is hyper-invertible if for any index sets $R \subseteq \{1, 2, ..., r\}$ and $C \subseteq \{1, 2, ..., c\}$ with $|R| = |C| > 0$, the matrix M_R^C is invertible, where M_R denotes the matrix consisting of the rows $i \in R$ of M, M^C denotes the matrix consisting of the columns $j \in C$ of M, and $M_R^C = (M_R)^C$.*

We point out a very useful property of hyper-invertible matrices, which is a more generalized version compared with that shown in [BTH08].

Lemma 1. *Let M be a hyper-invertible r-by-c matrix and $(y_1, ..., y_r) = M(x_1, ..., x_c)$. Then for any sets of indices $A \subseteq \{1, 2, ..., c\}$ and $B \subseteq \{1, 2, ..., r\}$ 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 $\{x_i\}_{i \notin A}, \{y_j\}_{j \notin B}$.*

3.5 Secret Sharing

In our protocol, we use the standard Shamir secret sharing scheme [Sha79]. We adopt the notion of d-shared in [BTH08].

Definition 2 ([BTH08]). *We say that a value s is (correctly) d-shared (among the parties in $\mathcal{P}_{\texttt{active}}$) if every honest party $P_i \in \mathcal{P}_{\texttt{active}}$ is holding a share s_i of s, such that there exists a degree-d polynomial $p(\cdot)$ with $p(0) = s$ and $p(\alpha_i) = s_i$ for every $P_i \in \mathcal{P}_{\texttt{active}}$. The vector $(s_1, s_2, ..., s_{n'})$ of shares is called a d-sharing of s, and is denoted by $[s]_d$. A (possibly incomplete) set of shares is called d-consistent if these shares lie on a degree-d polynomial.*

For every function $f : \mathbb{F}^m \to \mathbb{F}^{m'}$, by writing $f([x^{(1)}]_d, [x^{(2)}]_d, ..., [x^{(m)}]_d)$, we mean f is applied on $(x_i^{(1)}, x_i^{(2)}, ..., x_i^{(m)})$ for every $i \in \{1, 2, ..., n'\}$. Especially, when we say all parties in $\mathcal{P}_{\texttt{active}}$ locally compute $f([x^{(1)}]_d, [x^{(2)}]_d, ..., [x^{(m)}]_d)$, each party P_i computes $f(x_i^{(1)}, x_i^{(2)}, ..., x_i^{(m)})$.

We point out that Shamir secret sharing scheme is linear, i.e., for every two d-sharing $[u]_d, [v]_d$, $[c_1 u + c_2 v]_d = c_1[u]_d + c_2[v]_d$. We also heavily use the following two facts: in the case $t' < (n' - d)/2$, a d-sharing $[u]_d$ is correctable with at most t' errors, e.g., by Berlekamp-Welch Algorithm; in the case $t' < n' - d$, a d-sharing $[u]_d$ is detectable with at most t' errors, due to the fact that two different degree-d polynomials $f_1(\cdot), f_2(\cdot)$ over $\mathbb{F}$ can have at most d points where two polynomials are equal. Particularly, a t-sharing is always correctable in our setting since $n' - 2t' = T = n - 2t > t$ and therefore $t' < (n' - t)/2$.

We say a t-sharing $[v]_t$ is correct (or v is correctly t-shared) if all shares held by honest parties lie on a degree-t polynomial. As we mentioned above, a correct t-sharing is always recoverable.

In our protocol, we use $[s]_{d_1, d_2}$ to represents two sharings of the same value s, one is d_1-sharing and the other one is d_2-sharing. We use $[s]_{d_1, d_2, d_3}$ to represent three sharings of the same value s, d_1-sharing, d_2-sharing and d_3-sharing respectively.

3.6 Batched Reconstruction

We directly borrow the procedure from [BTH08] to reconstruct a batch of d-sharings. The procedure RECONS takes T d-sharings as input and reconstructs each sharing to all parties.

Procedure 2 RECONS($\mathcal{P}_{\texttt{active}}, d, [s^1]_d, [s^2]_d, \ldots, [s^T]_d$) [BTH08]

1: All parties agree on n' different values $\beta_1, \beta_2, \ldots, \beta_{n'} \in \mathbb{F}$.
2: Expansion:
For every $j \in \{1, 2, \ldots, n'\}$, all parties (locally) expand $[s^1]_d, [s^2]_d, \ldots, [s^T]_d$ into an error correction code $[u^1]_d, [u^2]_d, \ldots, [u^{n'}]_d$ as:

$$[u^j]_d = [s^1]_d + [s^2]_d\beta_j + [s^3]_d\beta_j^2 + \ldots + [s^T]_d\beta_j^{T-1}$$

3: Collecting shares of $[u^i]_d$:
For every party P_i, all other parties send their shares of $[u^i]_d$ to P_i.
4: Each party P_i tries to reconstruct u^i from the shares it received:
P_i checks whether there exists a degree-d polynomial f such that at least $\min\{d + t' + 1, n'\}$ of the shares lie on it. If not, P_i sets its happy-bit to `unhappy`. P_i sends the value $u^i = f(0)$ or $\perp$ (in the case that f does not exist) to other parties.
5: On receiving $u^1, \ldots, u^{n'}$, each party P_i tries to reconstruct and output $s^1, \ldots, s^T$:
If there exists a degree-$(T-1)$ polynomial g such that at least $T + t'$ values of $u^1, u^2, \ldots, u^{n'}$ lie on it, P_i computes $s^1, s^2, \ldots, s^T$ from any T of them. Otherwise, P_i sets its happy-bit to `unhappy` and sets $s^j = \perp$ for all $j \in \{1, \ldots, T\}$. P_i takes $s^1, \ldots, s^T$ as output.

We point out two facts about the procedure RECONS, which are shown in [BTH08].

1. If $d < T = n' - 2t'$ and all d-sharings are correct, then RECONS always successfully reconstructs the sharings to parties in $\mathcal{P}_{\texttt{active}}$. As we mentioned before, a d-sharing is correctable with at most t' errors when $t' < (n' - d)/2$, which is equivalent to $d < n' - 2t' = T$.
2. If $d < n' - t'$ and all d-sharings are correct, then either all sharings are correctly reconstructed or at least one happy-bit of an honest party is `unhappy`.

The procedure RECONS will reconstruct $T = \Omega(n)$ sharings while communicating $O(n^2)$ elements to collect shares of $[u^i]_d$ in Step 3 and distributed the reconstructed u^i to all parties. Thus, the overall communication complexity is $O(n^2\kappa)$ bits. Note that for each sharing, the communication complexity to reconstruct it is $O(n)$ elements in average.

3.7 Input Gates

We directly use the result in [BTH08] where they provided a protocol for input gates with communication complexity $O(c_I n\kappa + n^3\kappa)$ bits, where c_I is the number of input gates in the circuit. The formal functionality appears in $\mathcal{F}_{\text{input}}$.

Functionality 3 $\mathcal{F}_{\text{input}}(c_I)$

1: $\mathcal{F}_{\text{input}}$ receives inputs, which are denoted by $v^1, v^2, \ldots, v^{c_I} \in \mathbb{F}$, from all parties including honest and corrupted parties. $\mathcal{F}_{\text{input}}$ initially sets $\mathtt{state}_j = 1$ for $j \in \{1, \ldots, c_I\}$.

2: From $j = 1$ to c_I, $\mathcal{F}_{\text{input}}$ asks $\mathcal{S}$ what to do next:
   - On receiving $(P_i, P_k, \text{disputed})$ where $\mathcal{C}_{\mathtt{active}} \bigcap \{P_i, P_k\} \neq \emptyset$, $\mathcal{F}_{\text{input}}$ sets $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} \backslash \{P_i, P_k\}$ and $\mathcal{C}_{\mathtt{active}} := \mathcal{C}_{\mathtt{active}} \backslash \{P_i, P_k\}$.
   - On receiving $(v^j, \text{corrupted})$ where v^j is sent by a corrupted party, $\mathcal{F}_{\text{input}}$ sets $\mathtt{state}_j = 0, \mathcal{P}^j = \mathcal{P}_{\mathtt{active}}, \mathcal{C}^j = \mathcal{C}_{\mathtt{active}}$ and then handle $j := j + 1$.
   - On receiving $\{v_s^j\}_{P_s \in \mathcal{C}_{\mathtt{active}}}$, $\mathcal{F}_{\text{input}}$ sets $\mathcal{P}^j = \mathcal{P}_{\mathtt{active}}, \mathcal{C}^j = \mathcal{C}_{\mathtt{active}}$ and then handle $j := j + 1$.

3: For each $j \in \{1, \ldots, c_I\}$, if $\mathtt{state}_j = 1$, $\mathcal{F}_{\text{input}}$ computes a random t-sharing $[v^j]_t$ of the input v^j received in the first step, such that for all $P_s \in \mathcal{C}^j$, the s-th share of $[v^j]_t$ is v_s^j. If $\mathtt{state}_j = 0$, $\mathcal{F}_{\text{input}}$ sets $[v^j]_t = [0]_0$.

4: For every $j \in \{1, \ldots, c_I\}$ and $P_i \in \mathcal{P}^j$, $\mathcal{F}_{\text{input}}$ sends v_i^j to P_i. $\mathcal{F}_{\text{input}}$ also sends $\mathcal{P}_{\mathtt{active}}$ to all parties.

We refer the reader to the appendix of [BTH08] for the construction of a protocol which instantiates $\mathcal{F}_{\text{input}}$.

4 4-Consistency

In our protocol, we first use random $(n'-1)$-sharings to help evaluate the circuit. Indeed, there is no redundancy in a $(n'-1)$-sharing: to reconstruct the value, all shares from $\mathcal{P}_{\mathtt{active}}$ are needed. However, it makes the sharing vulnerable and the verification becomes much harder due to the lack of redundancy, e.g., every party is able to change the value by changing its own share without being detected.

Therefore, we need a tool to let each party commit their shares *after* evaluating the circuit to help verifying the honesty. To this end, we introduce the notion 4-consistency. Recall that t is the maximum number of corrupted parties an adversary can control and n' is the number of active parties.

Definition 3. *For a partition π of $\mathcal{P}_{\mathtt{active}} = \mathcal{P}_1 \bigcup \mathcal{P}_2 \bigcup \mathcal{P}_3$ such that $|P_1|, |P_2|, |P_3| \leq t + 1$, a tuple of t-sharings $[\![r]\!] = ([_0r]_t, [_1r]_t, [_2r]_t, [_3r]_t)$ is a 4-consistent tuple w.r.t. π if ${}_0r = r$ and there exists a degree-$(n'-1)$ polynomial $p(\cdot)$ with $p(0) = r$ and for all $P_i \in \mathcal{P}_j$, $p(\alpha_i)$ is the i-th share of the sharing $[_jr]_t$.*

In fact, the vector $(p(\alpha_1), p(\alpha_2), ..., p(\alpha_{n'}))$ is a $(n'-1)$-sharing of r by definition. We denote it as $[r]_{n'-1}$. In our construction, $[r]_{t,n'-1}$ is first generated to do evaluation. Then, in the verification step, $[_1r]_t, [_2r]_t, [_3r]_t$ are generated to commit the shares of $[r]_{n'-1}$. This is due to the fact that t-sharings are correctable (as we explained in Section 3.5). Therefore, each share of $[r]_{n'-1}$ can be recovered no matter how corrupted parties change their shares.

Lemma 2. *4-consistency is preserved under linear combinations*

Proof. We show that, for every two 4-consistent tuples $[\![r]\!], [\![s]\!]$ which are w.r.t. π and constants c_1, c_2,

$$\begin{aligned}[\![c_1 r + c_2 s]\!] &:= c_1[\![r]\!] + c_2[\![s]\!] \\ &= (c_1[{}_0r]_t + c_2[{}_0s]_t, c_1[{}_1r]_t + c_2[{}_1s]_t, c_1[{}_2r]_t + c_2[{}_2s]_t, c_1[{}_3r]_t + c_2[{}_3s]_t)\end{aligned}$$

is still 4-consistent w.r.t. π.

To see this, by the linearity of Shamir secret sharing scheme, each entry of the resulting tuple is still a t-sharing. Especially, $c_1[{}_0r]_t + c_2[{}_1s]_t = [c_1({}_0r) + c_2({}_0s)]_t$ and $c_1 r + c_2 s = c_1({}_0r) + c_2({}_0s)$.

Let $p(\cdot), q(\cdot)$ be the polynomials such that $p_1(0) = r, p_2(0) = s$ and for every $j \in \{1,2,3\}$ and $P_i \in \mathcal{P}_j$, $p_1(\alpha_i), p_2(\alpha_i)$ are the i-th shares of the sharings $[{}_jr]_t, [{}_js]_t$ respectively (as per the definition). Let $p_3 = c_1 p_1 + c_1 p_2$. Then $p_3(0) = c_1 p_1(0) + c_2 p_2(0) = c_1 r + c_2 s$. For every $P_i \in \mathcal{P}_j$, $p_3(\alpha_i) = c_1 p_1(\alpha_i) + c_2 p_2(\alpha_i)$ which is exactly the i-th share of $c_1[{}_jr]_t + c_2[{}_js]_t$. □

We say a 4-consistent tuple $[\![r]\!] = ([{}_0r]_t, [{}_1r]_t, [{}_2r]_t, [{}_3r]_t)$ is correct if 1) each of the t-sharings is correct and 2) after correcting possible wrong shares held by corrupted parties, it is 4-consistent.

5 Building Block

In this section, we introduce several building blocks which will be utilized in the full protocol.

5.1 Random Triple-Sharings

The following procedure, TripleShareRandom($\mathcal{P}_{\texttt{active}}, d_1, d_2, d_3$), is used to generate and distribute T random triple-sharings $\{[r^i]_{d_1,d_2,d_3}\}_{i=1}^T$ where $r^1, \ldots, r^T$ are sampled uniformly from $\mathbb{F}$ and $d_1, d_2, d_3 \geq t'$. It finally outputs either T valid random triple-sharings or a pair of disputed parties. The ideal functionality is described in $\mathcal{F}_{\text{triple}}$.

Functionality 4 $\mathcal{F}_{\text{triple}}(d_1, d_2, d_3)$

1: On receiving $(P_i, P_k, \text{disputed})$, where $\mathcal{C}_{\texttt{active}} \bigcap \{P_i, P_k\} \neq \emptyset$, from $\mathcal{S}$, $\mathcal{F}_{\text{triple}}$ sends $(P_i, P_k, \text{disputed})$ to all parties.

2: On receiving $(\{r_s^{1,d_1}, r_s^{1,d_2}, r_s^{1,d_3}\}_{P_s \in \mathcal{C}_{\texttt{active}}}, \ldots, \{r_s^{T,d_1}, r_s^{T,d_2}, r_s^{T,d_3}\}_{P_s \in \mathcal{C}_{\texttt{active}}})$ from $\mathcal{S}$, $\mathcal{F}_{\text{triple}}$ samples $r^1, r^2, \ldots, r^T$ uniformly from $\mathbb{F}$. Then $\mathcal{F}_{\text{triple}}$ randomly generates $[r^1]_{d_1,d_2,d_3}, \ldots, [r^T]_{d_1,d_2,d_3}$ such that for every $j \in \{1, \ldots, T\}$ and $P_s \in \mathcal{C}_{\texttt{active}}$, the s-th shares of $[r^j]_{d_1,d_2,d_3}$ are $r_s^{j,d_1}, r_s^{j,d_2}, r_s^{j,d_3}$ respectively. For every $j \in \{1, \ldots, T\}$ and $P_i \in \mathcal{P}_{\texttt{active}}$, $\mathcal{F}_{\text{triple}}$ sends the i-th shares of $[r^j]_{d_1,d_2,d_3}$ to P_i.

The basic construction is very similar to the protocol which generates random double-sharings in [BTH08]. The only difference is that we generate triple-sharings instead of double-sharings.

In the beginning of the computation, all parties will agree on a constant hyper-invertible matrix M of size $n' \times n'$, which will be employed to extract randomness. The first step of the protocol is to let each party P_i distribute a triple-sharing $[s^i]_{d_1,d_2,d_3}$ of a random value s^i. Then apply the hyper-invertible matrix M on them, i.e., $([r^1]_{d_1,d_2,d_3}, \ldots, [r^{n'}]_{d_1,d_2,d_3}) = M([s^1]_{d_1,d_2,d_3}, \ldots, [s^{n'}]_{d_1,d_2,d_3})$.

For the last $2t'$ triple-sharings, each of them is reconstructed by a different party. Each party who reconstructs one of the triple-sharings checks whether they are valid and sets its happy-bit to `unhappy` if the triple-sharing is invalid. Finally, all parties take the remaining $T = n' - 2t'$ triple-sharings as output.

It guarantees that either the output (i.e., $[r^1]_{d_1,d_2,d_3}, \ldots, [r^T]_{d_1,d_2,d_3}$) is correct or at least one happy-bit of an honest party is `unhappy`.

Formally,

Procedure 5 TripleShareRandom($\mathcal{P}_{\texttt{active}}, d_1, d_2, d_3$)

1: All parties agree on a hyper-invertible matrix M of size $n' \times n'$.
2: Parties distribute their own randomness:
Each party P_i samples $s^i \in \mathbb{F}$ uniformly. Then randomly generate $[s^i]_{d_1,d_2,d_3}$. For each other party P_j, P_i sends the j-th shares of $[s^i]_{d_1,d_2,d_3}$ to P_j.
3: Extracting randomness from honest parties:
All parties locally compute

$$([r^1]_{d_1,d_2,d_3}, \ldots, [r^{n'}]_{d_1,d_2,d_3}) = M([s^1]_{d_1,d_2,d_3}, \ldots, [s^{n'}]_{d_1,d_2,d_3})$$

4: Check the correctness:
1. For $j \in \{T+1, \ldots, n'\}$, all parties send their shares of $[r^j]_{d_1,d_2,d_3}$ to P_j.
2. P_j checks whether the triple-sharing it received is valid, i.e., all shares of $[r^j]_{d_1}, [r^j]_{d_2}, [r^j]_{d_3}$ lie on degree-d_1 ,degree-d_2, degree-d_3 polynomials g_1, g_2, g_3 respectively, and $g_1(0) = g_2(0) = g_3(0)$. If not, P_j sets its happy-bit to `unhappy`.
5: All parties take the first T triple-sharings $[r_1]_{d_1,d_2,d_3}, \ldots, [r_T]_{d_1,d_2,d_3}$ as output.

In procedure TripleShareRandom, parties communicate $O(n^2)$ elements to distribute randomness in Step 2 and check correctness in Step 4. Thus, the overall communication complexity is $O(n^2\kappa)$ bits. Note that the communication complexity for generating each random triple sharing is $O(n)$ elements.

Let TripleShareRandom-PE := Party-Elimination(TripleShareRandom). Then TripleShareRandom-PE securely computes $\mathcal{F}_{\text{triple}}$.

Lemma 3 ([BTH08]). *The protocol* TripleShareRandom-PE *computes* $\mathcal{F}_{triple}$ *with perfect security when* $|\mathcal{C}_{\texttt{active}}| < |\mathcal{P}_{\texttt{active}}|/3$.

The overall communication complexity of TripleShareRandom-PE is $O(n^2\kappa)$ bits.

5.2 Random Multiplication Tuples

The procedure GenerateTuples($\mathcal{P}_{\texttt{active}}$) is used to generate T correctly and independently random tuples $([a]_{t,n'-1}, [b]_{t,n'-1}, [c]_t)$, which we call multiplication tuples, where a, b are uniformly random and $c = ab$. It outputs either T random multiplication tuples or a pair of disputed parties. The ideal functionality is described in $\mathcal{F}_{\text{multi-tuple}}$.

Functionality 6 $\mathcal{F}_{\text{multi-tuple}}(\mathcal{P}_{\texttt{active}})$

1: On receiving $(P_i, P_k, \text{disputed})$, where $\mathcal{C}_{\texttt{active}} \bigcap \{P_i, P_k\} \neq \emptyset$, from $\mathcal{S}$, $\mathcal{F}_{\text{multi-tuple}}$ sends $(P_i, P_k, \text{disputed})$ to all parties.

2: On receiving

$$(\{a_s^{1,t}, a_s^{1,n'-1}, b_s^{1,t}, b_s^{1,n'-1}, c_s^{1,t}\}_{P_s \in \mathcal{C}_{\texttt{active}}}, \ldots, \{a_s^{T,t}, a_s^{T,n'-1}, b_s^{T,t}, b_s^{T,n'-1}, c_s^{T,t}\}_{P_s \in \mathcal{C}_{\texttt{active}}})$$

from $\mathcal{S}$, $\mathcal{F}_{\text{multi-tuple}}$ samples $a^1, \ldots, a^T, b^1, \ldots, b^T$ uniformly from $\mathbb{F}$ and computes $c^1 = a^1 b^1, \ldots, c^T = a^T b^T$. Then $\mathcal{F}_{\text{multi-tuple}}$ randomly generates

$$([a^1]_{t,n'-1}, [b^1]_{t,n'-1}, [c^1]_t), \ldots, ([a^T]_{t,n'-1}, [b^T]_{t,n'-1}, [c^T]_t)$$

such that for every $j \in \{1, \ldots, T\}$ and $P_s \in \mathcal{C}_{\texttt{active}}$, the s-th shares of $([a^j]_{t,n'-1}, [b^j]_{t,n'-1}, [c^j]_t)$ are $(a_s^{1,t}, a_s^{1,n'-1}, b_s^{1,t}, b_s^{1,n'-1}, c_s^{1,t})$ respectively. For every $j \in \{1, \ldots, T\}$ and $P_i \in \mathcal{P}_{\texttt{active}}$, $\mathcal{F}_{\text{multi-tuple}}$ sends the i-th shares of $([a^j]_{t,n'-1}, [b^j]_{t,n'-1}, [c^j]_t)$ to P_i.

The basic construction is very similar to the protocol which generates random triples in [BTH08]. The difference is that we generate triple-sharings $[a]_{t',t,n'-1}, [b]_{t',t,n'-1}, [r]_{t,2t',n'-1}$ instead of double-sharings $[a]_{t',t}, [b]_{t',t}, [r]_{t,2t'}$ in the beginning. However $[a]_{n'-1}, [b]_{n'-1}$ are directly output and $[r]_{n'-1}$ is discarded.

GenerateTuples($\mathcal{P}_{\texttt{active}}$) first invokes TripleShareRandom to generate random triple sharings $[a^1]_{t',t,n'-1}, \ldots, [a^T]_{t',t,n'-1}, [b^1]_{t',t,n'-1}, \ldots, [b^T]_{t',t,n'-1}$ and $[r^1]_{t,2t',n'-1}, \ldots, [r^T]_{t,2t',n'-1}$. For every $i \in \{1, \ldots, T\}$, $[a^i]_{t'}, [b^i]_{t'}$ are used to compute $[c^i]_{2t'} = [a^i]_{t'}[b^i]_{t'}$ locally and $[r^i]_{t,2t'}$ is used to generate $[c^i]_t$.

Procedure 7 GenerateTuples($\mathcal{P}_{\texttt{active}}$)

1: Generate random triple-sharings:
All parties invoke TripleShareRandom($\mathcal{P}_{\texttt{active}}, t', t, n'-1$) two times to generate $[a^1]_{t',t,n'-1}, \ldots, [a^T]_{t',t,n'-1}$ and $[b^1]_{t',t,n'-1}, \ldots, [b^T]_{t',t,n'-1}$. Then invoke TripleShareRandom($\mathcal{P}_{\texttt{active}}, t, 2t', n'-1$) to generate $[r^1]_{t,2t',n'-1}, \ldots, [r^T]_{t,2t',n'-1}$.

2: For $j \in \{1, \ldots, T\}$, all parties locally compute $[c^j]_{2t'} = [a^j]_{t'}[b^j]_{t'}$ where $c^j = a^j b^j$. Since each party directly multiplies its shares, the result is a $2t'$-sharing.

3: For $j \in \{1, \ldots, T\}$, the parties in $\mathcal{P}_{\texttt{active}}$ locally compute $[d^j]_{2t'} = [c^j]_{2t'} - [r^j]_{2t'}$.

4: Invoke RECONS($\mathcal{P}_{\texttt{active}}, 2t', [d^1]_{2t'}, \ldots, [d^T]_{2t'}$) to reconstruct $d^1, \ldots, d^T$.
5: For $j \in \{1, \ldots, T\}$, the parties in $\mathcal{P}_{\texttt{active}}$ locally compute $[c^j]_t = d^j + [r^j]_t$.
6: Output the T tuples $([a^1]_{t,n'-1}, [b^1]_{t,n'-1}, [c^1]_t), \ldots, ([a^T]_{t,n'-1}, [b^T]_{t,n'-1}, [c^T]_t)$.

As parties only communicate with each other when invoking TRIPLESHARERANDOM and RECONS, the communication complexity of GENERATETUPLES is thus $O(n^2\kappa)$ bits. Note that the communication cost of each random multiplication tuple is $O(n)$ elements.

Let GENERATETUPLES-PE := PARTY-ELIMINATION(GENERATETUPLES). Then GENERATETUPLES-PE securely computes $\mathcal{F}_{\text{multi-tuple}}$.

Lemma 4 ([BTH08]). *The protocol* GENERATETUPLES-PE *computes* $\mathcal{F}_{\textit{multi-tuple}}$ *with perfect security when* $|\mathcal{C}_{\texttt{active}}| < |\mathcal{P}_{\texttt{active}}|/3$.

The overall communication complexity of GENERATETUPLES-PE is $O(n^2\kappa)$ bits.

5.3 Generating 4-consistent Tuples

The procedure QUADRUPLESHARERANDOM($\mathcal{P}_{\texttt{active}}, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$) is used to generate T correct and random 4-consistent tuples $[\![r]\!] = ([_0r]_t, [_1r]_t, [_2r]_t, [_3r]_t)$. The procedure takes $\mathcal{P}_{\texttt{active}}$, and a partition $\mathcal{P}_{\texttt{active}} = \mathcal{P}_1 \bigcup \mathcal{P}_2 \bigcup \mathcal{P}_3$, where $|\mathcal{P}_1|, |\mathcal{P}_2|, |\mathcal{P}_3| \leq t+1$, as input. It outputs either T correct random 4-consistent tuples or a pair of disputed parties. The ideal functionality is described in $\mathcal{F}_{\text{4-consistency}}$.

Functionality 8 $\mathcal{F}_{\text{4-consistency}}(\mathcal{P}_{\texttt{active}}, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3)$

1: On receiving $(P_i, P_k, \text{disputed})$, where $\mathcal{C}_{\texttt{active}} \bigcap \{P_i, P_k\} \neq \emptyset$, from $\mathcal{S}$, $\mathcal{F}_{\text{4-consistency}}$ sends $(P_i, P_k, \text{disputed})$ to all parties.
2: On receiving $(\{_0r_s^1, _1r_s^1, _2r_s^1, _3r_s^1\}_{P_s \in \mathcal{C}_{\texttt{active}}}, \ldots, \{_0r_s^T, _1r_s^T, _2r_s^T, _3r_s^T\}_{P_s \in \mathcal{C}_{\texttt{active}}})$ from $\mathcal{S}$, $\mathcal{F}_{\text{4-consistency}}$ randomly generates

$$[\![r^1]\!], \ldots, [\![r^T]\!]$$

such that for every $j \in \{1, \ldots, T\}$ and $P_s \in \mathcal{C}_{\texttt{active}}$, the s-th shares of $[\![r^j]\!]$ are $(_0r_s^j, _1r_s^j, _2r_s^j, _3r_s^j)$ respectively. For every $j \in \{1, \ldots, T\}$ and $P_i \in \mathcal{P}_{\texttt{active}}$, $\mathcal{F}_{\text{4-consistency}}$ sends the i-th shares of $[\![r^j]\!]$ to P_i.

The construction of QUADRUPLESHARERANDOM is similar to TRIPLESHARERANDOM by the following means: First, each party deals a random 4-consistent tuple, and then a hyper-invertible matrix is applied to extract the randomness. For the last $2t'$ out of n' 4-consistent tuples, they will then be reconstructed to check whether corrupted parties cheated in the computation. Finally, the remaining $n' - 2t' = T$ 4-consistent tuples will be output.

Procedure 9 QUADRUPLESHARERANDOM($\mathcal{P}_{\texttt{active}}, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$)

1: All parties agree on a hyper-invertible matrix M.
2: Parties distribute their own randomness:
Each party P_i generates a random 4-consistent tuple $[\![s^i]\!] = ([_0s^i]_t, [_1s^i]_t, [_2s^i]_t, [_3s^i]_t)$. For each other party P_j, P_i sends the j-th shares of $[\![s^i]\!]$ to P_j.
3: Extracting randomness from honest parties:
All parties locally compute

$$([\![r^1]\!], \ldots, [\![r^{n'}]\!]) = M([\![s^1]\!], \ldots, [\![s^{n'}]\!]).$$

4: Check the correctness:
1. For $j \in \{T+1, \ldots, n'\}$, all parties send their shares of $[\![r^j]\!]$ to P_j.
2. P_j checks whether the 4-consistent tuple $[\![r^j]\!]$ is valid. If not, P_j sets its happy-bit to `unhappy`.
5: All parties take the first T tuples $[\![r^1]\!], \ldots, [\![r^T]\!]$ as output.

Parties communicate $O(n^2)$ elements to deal n' and reconstruct $2t'$ 4-consistent tuples, so the overall communication complexity is $O(n^2\kappa)$ bits. Note that in average, the communication cost for each 4-consistent tuple is $O(n)$ elements.

Let QuadrupleShareRandom-PE := Party-Elimination(QuadrupleShareRandom). Then QuadrupleShareRandom-PE securely computes $\mathcal{F}_{\text{4-consistency}}$.

Lemma 5. *The protocol* QuadrupleShareRandom-PE *computes* $\mathcal{F}_{\text{4-consistency}}$ *with perfect security when* $|\mathcal{C}_{\texttt{active}}| < |\mathcal{P}_{\texttt{active}}|/3$.

The overall communication complexity of QuadrupleShareRandom-PE is $O(n^2\kappa)$ bits.

5.4 Random 0-Sharings

The protocol ZeroShareRandom is used to generate T random t-sharings of 0. It outputs either T correct and random t-sharings of 0 or a pair of disputed parties. The ideal functionality is described in $\mathcal{F}_{\text{zero}}$.

Functionality 10 $\mathcal{F}_{\text{zero}}(\mathcal{P}_{\texttt{active}})$

1: On receiving $(P_i, P_k, \text{disputed})$, where $\mathcal{C}_{\texttt{active}} \bigcap \{P_i, P_k\} \neq \emptyset$, from $\mathcal{S}$, $\mathcal{F}_{\text{zero}}$ sends $(P_i, P_k, \text{disputed})$ to all parties.
2: On receiving $(\{r_s^1\}_{P_s \in \mathcal{C}_{\texttt{active}}}, \ldots, \{r_s^T\}_{P_s \in \mathcal{C}_{\texttt{active}}})$ from $\mathcal{S}$, $\mathcal{F}_{\text{zero}}$ randomly generates $[0^1]_t, \ldots, [0^T]_t$ such that for all $j \in \{1, \ldots, T\}$ and $P_s \in \mathcal{C}_{\texttt{active}}$, the s-th share of $[0^j]_t$ is r_s^j. For all $j \in \{1, \ldots, T\}$ and $P_i \in \mathcal{P}_{\texttt{active}}$, $\mathcal{F}_{\text{4-consistency}}$ sends the i-th shares of $[0^1]_t, \ldots, [0^T]_t$ to P_i.

ZeroShareRandom first invokes $\mathcal{F}_{\text{triple}}(t, t, t)$ to generate T random triple-sharings $[r]_{t,t,t}$. Then computes $[0]_t$ by subtracting the first t-sharing of r from the second t-sharing of r. Formally,

Protocol 11 ZeroShareRandom($\mathcal{P}_{\texttt{active}}$)

1: Generate random triple-sharings:
   All parties invoke $\mathcal{F}_{\text{triple}}(t,t,t)$ to generate $[r^1]_{t,t,t},\ldots,[r^T]_{t,t,t}$. If $\mathcal{F}_{\text{triple}}$ outputs $(P_i,P_k,\text{disputed})$, all parties halt. We write each triple-sharings $[r^j]_{t,t,t}$ as $([a^j]_t,[b^j]_t,[c^j]_t)$ where $r^j=a^j=b^j=c^j$ to distinguish these three t-sharings of the same value r^j.
2: For $j\in\{1,\ldots,T\}$, all parties locally compute $[0^j]_t=[b^j]_t-[a^j]_t$. All parties take $[0^1]_t,\ldots,[0^T]_t$ as output.

As we invoke $\mathcal{F}_{\text{triple}}(t,t,t)$ to generate T random sharings of 0 from T random triple-sharings, the overall communication complexity is $O(n^2\kappa)$ bits. And in average, the communication cost for each random t-sharing of 0 is $O(n)$ elements.

5.5 Check Consistency

The procedure CheckConsistency is used to check whether a party $P_{\texttt{king}}$ sent T same elements to all other parties. It outputs either $\perp$ or a disputed pairs.

We may think $P_{\texttt{king}}$ distributes T 0-sharings (which are essentially constant values) to all other parties. Suppose these sharings are $[d^1]_0,\ldots,[d^T]_0$. In the beginning, all parties agree on a hyper-invertible matrix M of size $(T+t')\times T$. Then all parties (locally) compute $([r^1]_0,\ldots,[r^{T+t'}]_0)=M([d^1]_0,\ldots,[d^T]_0)$. Each 0-sharing is reconstructed by a different party and each party who reconstructs one of the 0-sharing checks whether it is valid and sets its happy-bit to `unhappy` if not. Note that at least T sharings are checked by honest parties. If all honest parties are satisfied with the execution, by the property of hyper-invertible matrices, all honest parties received the same T elements from $P_{\texttt{king}}$.

However, in the fault-location phase, just providing all information in the computation phase is not enough. To find a pair of disputed parties, $P_{\texttt{king}}$ should send these T elements to the referee. Formally,

---

Procedure 12 CheckConsistency$(\mathcal{P}_{\texttt{active}},P_{\texttt{king}},[d^1]_0,\ldots,[d^T]_0)$

1: Initialization Phase:
   All parties initially set their happy-bits to `happy`.
2: Computation Phase:
   1. All parties agree on a hyper-invertible matrix M of size $(T+t')\times T$.
   2. All parties locally compute $([r^1]_0,\ldots,[r^{T+t'}]_0)=M([d^1]_0,\ldots,[d^T]_0)$.
   3. For $j\in\{1,\ldots,T+t'\}$, all parties send their shares of $[r^j]_0$ to P_j.
   4. P_j checks whether $[r^j]_0$ it receives is valid, i.e., all shares of $[r^j]_0$ are equal. If not, P_j sets its happy-bit to `unhappy`.
3: Fault Detection Phase:
   1. Each party sends their happy-bit to all other parties.
   2. For each party, if at least one of the happy-bits it receives is `unhappy`, set its happy-bit to be `unhappy`.
   3. All parties run a consensus protocol on their respective happy-bits. If the result is `happy`, all parties halt. Otherwise, run the following steps.
4: Fault Localization Phase:

1. All parties agree on a referee $P_r \in \mathcal{P}_{\texttt{active}}$. Every other party sends everything it generated, sent and received in the Computation Phase and Fault Detection Phase to P_r. $P_{\texttt{king}}$ sends $d^1, \ldots, d^T$ to P_r.
2. On receiving all information from other parties, P_r simulates the Computation Phase and Fault Detection Phase. P_r broadcasts either $(P_i, \text{corrupt})$ (in the case P_i does not follow the procedure) or $(\ell, P_i, P_k, v, v', \text{disputed})$ where ℓ is the index of the message where P_i should have sent v to P_k while P_k claimed to have received $v' \neq v$.
   (a) If $(P_i, \text{corrupt})$ is broadcast, all parties set $E = \{P_r, P_i\}$.
   (b) Otherwise, P_i and P_k broadcast whether they agree with P_r. If P_i disagrees, set $E = \{P_r, P_i\}$; if P_k disagrees, set $E = \{P_r, P_k\}$; otherwise, set $E = \{P_i, P_k\}$.
3. All parties take E as output and halt.

In the computation phase of CheckConsistency, all parties send $O(n^2)$ elements. The remaining step is the same as Party-Elimination except that $P_{\texttt{king}}$ needs to send additional $O(n)$ elements to P_r in the fault localization phase. The overall communication complexity is $O(n^2\kappa)$ bits.

5.6 Check 4-Consistency

The procedure Check4Consistency is used to check whether each party distributed a correct 4-consistent tuple. The privacy is preserved when invoking this procedure. It outputs either $\perp$ or a pair of disputed parties.

In the beginning, all parties agree on a hyper-invertible matrix M of size $(T+t') \times T$. Then, all parties invoke $\mathcal{F}_{\text{4-consistency}}$ several times to generate enough number of random tuples of 4-consistent sharings. Each random 4-consistent tuple is associated to one input 4-consistent tuple and it is reconstructed to the dealer of the input tuple. Instead of checking the original one, we will check the summation of these two tuples.

Every time, up to T tuples are checked. All parties locally apply M on these T tuples to get $T + t'$ tuples. Each tuple is then reconstructed by a different party. Each party who reconstructs the tuple of sharings checks whether it is 4-consistent and sets its happy-bit to `unhappy` if not. Note that at least T tuples are checked by honest parties. If all honest parties are satisfied with the execution, by the property of hyper-invertible matrices and the linearity of 4-consistent sharings, these T tuples are correct and 4-consistent.

However, in the fault-location phase, the dealer cannot provide the original tuple of sharings to the referee. Instead, the dealer provides the new tuple which is the summation of the original one and a random one. Note that the original tuple is generated by the dealer which should be 4-consistent and the random tuple is 4-consistent guaranteed by $\mathcal{F}_{\text{4-consistency}}$.

Procedure 13 Check4Consistency$(\mathcal{P}_{\texttt{active}}, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \{[\![s^j]\!]\}_{j=1}^{n'})$

1: Initialization Phase:

All parties initially set their happy-bits to `happy`.

2: Pre-Computation Phase:

1. All parties agree on a hyper-invertible matrix M of size $(T + t') \times T$.
2. From $j = 1$ to $\lceil n'/T \rceil$:
   All parties call $\mathcal{F}_{\text{4-consistency}}$ to generate T random tuples of 4-consistent sharings $[\![r^{T(j-1)+1}]\!], \ldots, [\![r^{T(j-1)+T}]\!]$. If $\mathcal{F}_{\text{4-consistency}}$ outputs $(P_i, P_k, \text{disputed})$, all parties take $(P_i, P_k, \text{disputed})$ as output and halt.
3. For $j \in \{1, \ldots, n'\}$, all parties send their shares of $[\![r^j]\!]$ to P_j.
4. For $j \in \{1, \ldots, n'\}$, all parties compute
$$[\![u^j]\!] := [\![s^j]\!] + [\![r^j]\!]$$

3: Computation Phase:

For $l > n'$, we set $[\![u^l]\!] := ([0]_0, [0]_0, [0]_0, [0]_0)$. For $j \in \{1, \ldots \lceil n'/T \rceil\}$:

1. All parties locally compute $([\![v^1]\!], \ldots, [\![v^{T+t'}]\!]) = M([\![u^{T(j-1)+1}]\!], \ldots, [\![u^{T(j-1)+T}]\!])$.
2. For $k \in \{1, \ldots, T + t'\}$, all parties send their shares of $[\![v^k]\!]$ to P_k.
3. P_k checks whether the 4-consistent tuple it received is valid. If not, P_k sets its happy-bit to `unhappy`.

4: Fault Detection Phase:

1. Each party sends their happy-bit to all other parties.
2. For each party, if at least one of the happy-bits it receives is `unhappy`, set its happy-bit to be `unhappy`.
3. All parties run a consensus protocol on their respective happy-bits. If the result is `happy`, all parties halt. Otherwise, run the following steps.

5: Fault Localization Phase:

1. All parties agree on a referee $P_r \in \mathcal{P}_{\texttt{active}}$. Every other party sends everything it generated, sent and received in the Computation Phase and Fault Detection Phase to P_r. Each party P_i also sends $[\![u^i]\!]$ to P_r.
2. On receiving all information from other parties, P_r simulates the Computation Phase and Fault Detection Phase. P_r broadcasts either $(P_i, \text{corrupt})$ (in the case P_i does not follow the procedure) or $(\ell, P_i, P_k, v, v', \text{disputed})$ where ℓ is the index of the message where P_i should have sent v to P_k while P_k claimed to have received $v' \neq v$.
   (a) If $(P_i, \text{corrupt})$ is broadcast, all parties set $E = \{P_r, P_i\}$.
   (b) Otherwise, P_i and P_k broadcast whether they agree with P_r. If P_i disagrees, set $E = \{P_r, P_i\}$; if P_k disagrees, set $E = \{P_r, P_k\}$; otherwise, set $E = \{P_i, P_k\}$.
3. All parties take E as output and halt.

In the pre-computation phase of Check4Consistency, all parties invoke $\mathcal{F}_{\text{4-consistency}}$ $\lceil n'/T \rceil = O(1)$ times and send $O(n^2)$ elements to reconstruct n' $[\![r^j]\!]$-s to different parties. In total, $O(n^2)$ elements are sent.

In the computation phase, all parties send $O(n^2)$ elements to reconstruct $T + t'$ $[\![v^k]\!]$-s to different parties each round and $\lceil n'/T \rceil = O(1)$ rounds are executed. The remaining step is the same as Party-Elimination except that, in the fault localization phase, each party P_i should send $[\![u^i]\!]$-s to P_r, which contains $O(n^2)$ elements in total. The overall communication complexity is $O(n^2\kappa)$ bits.

6 Protocol

In this section, we formally describe our construction. The main protocol is divided into several parts. In the first part, all input gates are handled by $\mathcal{F}_{\text{input}}$. We refer the reader to Section 3.7. The second part generates random shares for all random gates. In the third part, the circuit is divided into segments where each segment contains exactly T multiplication gates. Then each segment is evaluated sequentially. The last part handles the output gates.

6.1 Random Gates

The functionality $\mathcal{F}_{\text{rand}}$ is used to generate random sharings of uniform elements in $\mathbb{F}$. We use c_R for the number of random gates in the circuit.

Functionality 14 $\mathcal{F}_{\text{rand}}(c_R)$

1: From $j = 1$ to c_R, $\mathcal{F}_{\text{rand}}$ asks $\mathcal{S}$ what to do next:
   - On receiving $(P_i, P_k, \text{disputed})$ where $\mathcal{C}_{\texttt{active}} \bigcap \{P_i, P_k\} \neq \emptyset$, $\mathcal{F}_{\text{rand}}$ sets $\mathcal{P}_{\texttt{active}} := \mathcal{P}_{\texttt{active}} \backslash \{P_i, P_k\}$ and $\mathcal{C}_{\texttt{active}} := \mathcal{C}_{\texttt{active}} \backslash \{P_i, P_k\}$.
   - On receiving $\{v_s^j\}_{P_s \in \mathcal{C}_{\texttt{active}}}$, $\mathcal{F}_{\text{rand}}$ sets $\mathcal{P}^j = \mathcal{P}_{\texttt{active}}, \mathcal{C}^j = \mathcal{C}_{\texttt{active}}$ and continue to handle $j := j + 1$.

2: For every $j \in \{1, \ldots, c_R\}$, $\mathcal{F}_{\text{rand}}$ generates a random value $v^j \in \mathbb{F}$ and computes a random t-sharing $[v^j]_t$ such that for all $P_s \in \mathcal{C}^j$, the s-th share of $[v^j]_t$ is v_s^j.

3: For every $j \in \{1, \ldots, c_R\}$ and $P_i \in \mathcal{P}^j$, $\mathcal{F}_{\text{rand}}$ sends v_i^j to P_i. $\mathcal{F}_{\text{rand}}$ also sends $\mathcal{P}_{\texttt{active}}$ to all parties.

The formal instantiation of $\mathcal{F}_{\text{rand}}$ is described below.

Protocol 15 RAND$(\mathcal{P}_{\texttt{active}}, c_R)$

1: From $j = 1$ to $\lceil c_R/T \rceil$, do the follows:
   1. All parties in $\mathcal{P}_{\texttt{active}}$ call $\mathcal{F}_{\text{triple}}(n' - 1, t, t')$.
   2. If $\mathcal{F}_{\text{triple}}$ outputs $(P_i, P_k, \text{disputed})$, all parties set $\mathcal{P}_{\texttt{active}} := \mathcal{P}_{\texttt{active}} \backslash \{P_i, P_k\}$ and P_i, P_k halt. Repeat this step.
   3. Otherwise, all $(n' - 1)$-sharings and t'-sharings are discarded. Denote the T t-sharings as $[v^{T(j-1)+1}]_t, \ldots, [v^{T(j-1)+T}]_t$. All parties in $\mathcal{P}_{\texttt{active}}$ continue to handle $j := j + 1$.

2: All parties take $[v^1]_t, \ldots, [v^{c_R}]_t$ as output and the remaining sharings are discarded.

We now show that RAND securely computes $\mathcal{F}_{\text{rand}}$.

Lemma 6. *The protocol* RAND *computes $\mathcal{F}_{rand}$ with perfect security when $|\mathcal{C}_{\texttt{active}}| < |\mathcal{P}_{\texttt{active}}|/3$.*

Proof. Let $\mathcal{A}$ be the adversary in the real world. We show the existence of $\mathcal{S}$:
From $j = 1$ to $\lceil c_R/T \rceil$, $\mathcal{S}$ does the follows:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\text{triple}}(n'-1, t, t')$.
2. On receiving $(P_i, P_k, \text{disputed})$, where $\mathcal{C}_{\texttt{active}} \bigcap \{P_i, P_k\} \neq \emptyset$, from $\mathcal{A}$, $\mathcal{S}$ sets $\mathcal{P}_{\texttt{active}} := \mathcal{P}_{\texttt{active}} \backslash \{P_i, P_k\}$ and $\mathcal{C}_{\texttt{active}} := \mathcal{C}_{\texttt{active}} \backslash \{P_i, P_k\}$. $\mathcal{S}$ sends $(P_i, P_k, \text{disputed})$ to $\mathcal{F}_{\text{rand}}$. Repeat the loop from the beginning.
3. On receiving $(\{r_s^{1,n'-1}, r_s^{1,t}, r_s^{1,t'}\}_{P_s \in \mathcal{C}_{\texttt{active}}}, \{r_s^{T,n'-1}, r_s^{T,t}, r_s^{T,t'}\}_{P_s \in \mathcal{C}_{\texttt{active}}})$ from $\mathcal{A}$, for $i = 1, 2, \ldots, T$, $\mathcal{S}$ sets $\{v_s^{T(j-1)+i}\}_{P_s \in \mathcal{C}_{\texttt{active}}} = \{r_s^{i,t}\}_{P_s \in \mathcal{C}_{\texttt{active}}}$.
4. From $i = 1$ to T, if $T(j-1)+i \leq T$, $\mathcal{S}$ sends $\{v_s^{T(j-1)+i}\}_{P_s \in \mathcal{C}_{\texttt{active}}}$ to $\mathcal{F}_{\text{rand}}$.

Note that $\mathcal{S}$ does not send any message to $\mathcal{A}$. The view of $\mathcal{A}$ in either world is just empty. □

Note that, each time we repeat Step 1.2, at least one corrupted party is removed from $\mathcal{P}_{\texttt{active}}$. Thus, we will repeat Step 1.2 at most $t = O(n)$ times. Therefore, $\mathcal{F}_{\text{triple}}$ is invoked at most $\lceil c_R/T \rceil + O(n)$ times.

By using TripleShareRandom-PE to instantiate $\mathcal{F}_{\text{triple}}$ in Rand, the overall communication complexity is $O((\lceil c_R/T \rceil + O(n))n^2\kappa) = O(c_R n\kappa + n^3\kappa)$ bits.

6.2 Addition and Multiplication Gates

The circuit is first divided into several segments such that each segment `seg` contains T multiplication gates. All segments are evaluated sequentially. If a segment is evaluated successfully, then in the end, every output wire of this segment is a correct t-sharing. Otherwise, a pair of disputed parties is recognized. We first describe the procedure for evaluating one segment.

Procedure 16 Eval($\mathcal{P}_{\texttt{active}}$, `seg`)

1: Initialization:
All parties agree on a party $P_{\texttt{king}}$ and a partition of $\mathcal{P}_{\texttt{active}} = \mathcal{P}_1 \bigcup \mathcal{P}_2 \bigcup \mathcal{P}_3$ such that $|\mathcal{P}_1|, |\mathcal{P}_2|, |\mathcal{P}_3| \leq t+1$.
2: Generate multiplication tuples:
All parties invoke GenerateTuples-PE($\mathcal{P}_{\texttt{active}}$). If the result is $(P_i, P_k, \text{disputed})$, all parties take it as output and halt. Otherwise, run the following steps.
3: Evaluate `seg`:
For every addition gate, all parties apply addition on their own shares.
For every multiplication gate, a multiplication tuple generated in the first step is associated with it. We use $[x]_t, [y]_t$ for the input wires and $([a]_{t,n'-1}, [b]_{t,n'-1}, [c]_t)$ for the multiplication tuple.
   1. All parties compute $[d]_{n'-1} := [x]_t + [a]_{n'-1}$ and $[e]_{n'-1} := [y]_t + [b]_{n'-1}$.
   2. All parties send their shares of $[d]_{n'-1}$ and $[e]_{n'-1}$ to $P_{\texttt{king}}$.
   3. $P_{\texttt{king}}$ reconstructs the d and e. Then send these two elements back to all other parties.
   4. All parties compute $[z]_t := de - d[b]_t - e[a]_t + [c]_t$.
4: Check the consistency of $P_{\texttt{king}}$:
Let $d^1, \ldots, d^T, e^1, \ldots, e^T$ be the elements $P_{\texttt{king}}$ distributed in the last step. We view that step as $P_{\texttt{king}}$ distributing $[d^1]_0, \ldots, [d^T]_0, [e^1]_0, \ldots, [e^T]_0$.

1. All parties invoke the procedure CheckConsistency($\mathcal{P}_{\mathtt{active}}, P_{\mathtt{king}}, [d^1]_0, \ldots, [d^T]_0$). If the result is $(P_i, P_k, \text{disputed})$, all parties take it as output and halt.
2. All parties invoke the procedure CheckConsistency($\mathcal{P}_{\mathtt{active}}, P_{\mathtt{king}}, [e^1]_0, \ldots, [e^T]_0$). If the result is $(P_i, P_k, \text{disputed})$, all parties take it as output and halt.

5: Recompute all reconstructions:

We use $([x^1]_t, [y^1]_t), \ldots, ([x^T]_t, [y^T]_t)$ for the input wires of the multiplication gates in `seg` and $([a^1]_{t,n'-1}, [b^1]_{t,n'-1}, [c^1]_t), \ldots, ([a^T]_{t,n'-1}, [b^T]_{t,n'-1}, [c^T]_t)$ for the multiplication tuples associated with the multiplication gates.

1. For $j \in \{1, \ldots, T\}$, all parties compute $[d^j]_t = [x^j]_t + [a^j]_t, [e^j]_t = [y^j]_t + [b^j]_t$.
2. Invoke Recons($\mathcal{P}_{\mathtt{active}}, t, [d^1]_t, \ldots, [d^T]_t$) and Recons($\mathcal{P}_{\mathtt{active}}, t, [e^1]_t, \ldots, [e^T]_t$).
3. On receiving $d^1, \ldots, d^T, e^1, \ldots, e^T$, each party checks that whether they are correctly reconstructed by $P_{\mathtt{king}}$ in step 4. If they are all correct, take the shares of each output wires of `seg` as output and halt. Otherwise, find the first value which is incorrect. Without loss of generality, suppose $d^{i^\star}$ is the first incorrect value. Then do the following check.

6: Commit randomness used in GenerateTuples-PE:

For $P_i \in \mathcal{P}_{\mathtt{active}}$, let $[s^i]_{t,n'-1}$ denote the t-sharing and $(n'-1)$-sharing of s^i that P_i distributed in TripleShareRandom (which is used in GenerateTuples-PE).

1. For party P_i, it randomly generates $[_1 s^i]_t, [_2 s^i]_t, [_3 s^i]_t$ such that for $j \in \{1,2,3\}$ and $P_k \in \mathcal{P}_j$, the k-th share of $[_j s^i]_t$ (i.e., $_j s^i_k$) is the same as that of $[s^i]_{n'-1}$.
2. For $j \in \{1,2,3\}$ and $P_k \in \mathcal{P}_{\mathtt{active}} \backslash \mathcal{P}_j$, P_i sends the k-th share of $[_j s^i]_t$ to P_k.

7: Check 4-Consistency:

Let $[_0 s^i]_t := [s^i]_t$ and $[\![s^i]\!]$ denote the tuple of sharings $([_0 s^i]_t, [_1 s^i]_t, [_2 s^i]_t, [_3 s^i]_t)$.

1. All parties invoke the procedure

$$\text{Check4Consistency}(\mathcal{P}_{\mathtt{active}}, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \{[\![s^j]\!]\}_{j=1}^{n'})$$

2. If the result is $(P_i, P_k, \text{disputed})$, all parties take it as output and halt. Otherwise, run the following steps.

8: Find a disputed pair of parties:

Let M be the invertible matrix used in TripleShareRandom. Let $M_{i^\star}$ be the $i^\star$-th row of M. Then $[a^{i^\star}]_{n'-1} = M_{i^\star}([s^1]_{n'-1}, \ldots, [s^{n'}]_{n'-1})$.

1. For $j \in \{1,2,3\}$, all parties compute

$$[_j d^{i^\star}]_t = [x^{i^\star}]_t + M_{i^\star}([_j s^1]_t, \ldots, [_j s^{n'}]_t)$$

2. For $j \in \{1,2,3\}$, all parties send their shares of $[_j d^{i^\star}]_t$ to $P_{\mathtt{king}}$.
3. $P_{\mathtt{king}}$ finds $j^\star$ and $k^\star$ where the $k^\star$-th share of $[_{j^\star} d^{i^\star}]_t$ is not the value he received from $P_{k^\star}$ in Step 3. $P_{\mathtt{king}}$ broadcasts $(k^\star, \text{corrupt})$.
4. All parties take $\{P_{\mathtt{king}}, P_{k^\star}, \text{disputed}\}$ as output and halt.

Now, we analyze the correctness of Eval. The first two steps are straightforward. For Step 3, every addition gate can be computed locally by all parties. To

evaluate a multiplication gate, we use a random multiplication tuple. Instead of using random t-sharings in [BTH08], we use random $(n'-1)$-sharings (namely $[a]_{n'-1}$ and $[b]_{n'-1}$) to hide the t-sharings (namely $[x]_t$ and $[y]_t$). In this way, the messages that $P_{\texttt{king}}$ receives from honest parties are uniformly random. It prevents a malicious $P_{\texttt{king}}$ to gain addition knowledge from the shares of honest parties. Indeed, if all parties behave honestly, then all parties will get a random t-sharing $[z]_t$ where $z = xy$.

However, a corrupted party may send an incorrect share to $P_{\texttt{king}}$ or a malicious $P_{\texttt{king}}$ may send incorrect values back to all other parties. To detect such malicious behaviors, we first check whether $P_{\texttt{king}}$ sent the same values in Step 4. It is vital since it directly decides whether the shares of $[z]_t$ held by honest parties are consistent or not.

If all parties confirm that $P_{\texttt{king}}$ sent the same values to all other parties (at least to all honest parties), the next step is to check whether these reconstructed values are correct. This time, all parties use $[a]_t$ and $[b]_t$ instead of $[a]_{n'-1}$ and $[b]_{n'-1}$. Note that for each input wire of multiplication gates, all parties have already held a *correct* t-sharings, which is guaranteed by Step 4. Thus, we can reconstruct all t-sharings $[a]_t+[x]_t$ and $[b]_t+[y]_t$ for multiple layers of multiplication gates, which were evaluated in this segment, in parallel. If all reconstructions are the same as those in Step 3, then we confirm that this segment is evaluated successfully.

If all parties find at least one of the reconstructions is incorrect, then there must be some corrupted party which doesn't follow the protocol. All parties focus on the first incorrect one. Without loss of generality, we assume it is $d^{i^\star}$.

The main difficulty is that the redundancy is not enough to identify a pair of disputed parties. Therefore, in Step 6, all parties commit their randomness used in generating $[a^{i^\star}]_{t',t,n'-1}$, namely $[s^1]_{t',t,n'-1}, \ldots [s^{n'}]_{t',t,n'-1}$. Note that correct t'-sharings and t-sharings have already had enough redundancy in the sense that all parties can correct all shares no matter how corrupted parties change their shares. Therefore, we require that each party P_i commits $[s^i]_{n'-1}$ by using several t-sharings $[_1s^i]_t, [_2s^i]_t, [_3s^i]_t$. Together with $[_0s^i]_t = [s^i]_t$, the tuple of these 4 sharings forms a 4-consistent tuple.

In Step 7, all parties check whether these tuples are 4-consistent.

In the last step, for every $j \in \{1, 2, 3\}$, all parties compute

$$[_jd^{i^\star}]_t = [x^{i^\star}]_t + M_{i^\star}([_js^1]_t, \ldots, [_js^{n'}]_t)$$

Note that for $P_i \in \mathcal{P}_j$, the i-th share of $[_jd^{i^\star}]_t$ is exactly the share P_i should have sent to $P_{\texttt{king}}$ and this time, P_i cannot change its share without being caught. $P_{\texttt{king}}$ collects all shares of $[_1d^{i^\star}]_t, [_2d^{i^\star}]_t, [_3d^{i^\star}]_t$ and is able to broadcast a corrupted party. All party then view $P_{\texttt{king}}$ and the party it broadcast as a pair of disputed parties.

Now we analyze the communication complexity of Eval.

In Step 2, GenerateTuples-PE is invoked one time and the communication complexity is $O(n^2\kappa)$ bits. In Step 3, for each multiplication gate, $P_{\texttt{king}}$ receives from and sends to other parties $O(n)$ elements in total, which costs $O(n\kappa)$ bits.

In Step 4, CheckConsistency is invoked two times and the communication complexity is $O(n^2\kappa)$ bits. In Step 5, Recons is invoked two times and the communication complexity is $O(n^2\kappa)$ bits. In Step 6, all parties send $O(n^2)$ elements to distribute $[{}_1s^i]_t, [{}_2s^i]_t, [{}_3s^i]_t$. In Step 7, Check4Consistency is invoked one time and the communication complexity is $O(n^2\kappa)$ bits. In Step 8, all parties send $O(n)$ elements to $P_{\texttt{king}}$ to reconstruct $[{}_1d^{i^\star}]_t, [{}_2d^{i^\star}]_t, [{}_3d^{i^\star}]_t$ to $P_{\texttt{king}}$. Then another $O(n^2)$ elements are sent to let $P_{\texttt{king}}$ broadcast a corrupted party.

Therefore, the overall complexity is $O(n^2\kappa)$ bits.

6.3 Output Gates

The procedure Output helps reconstruct t-sharings to the parties specified by the output gates under the guarantee that, for each sharing associated with the output gates, the shares held by parties in $\mathcal{P}_{\texttt{active}} \backslash \mathcal{C}_{\texttt{active}}$ are consistent.

Procedure 17 Output($\mathcal{P}_{\texttt{active}}$)

All output gates are divided into several segments of size T. All segments are executed *sequentially*. For each segment:

1: All parties repeat the following steps until success:
   1. All parties call $\mathcal{F}_{\text{zero}}$.
   2. If $\mathcal{F}_{\text{zero}}$ outputs $(P_i, P_k, \text{disputed})$, all parties set $\mathcal{P}_{\texttt{active}} := \mathcal{P}_{\texttt{active}} \backslash \{P_i, P_k\}$. Redo the loop.
   3. If $\mathcal{F}_{\text{zero}}$ outputs T t-sharings of 0, break the loop.
2: Each output gate consumes one $[0]_t$ generated in the last step. For each output gate, $[s]_t$ denotes the t-sharing and $P_{i^\star} \in \mathcal{P}$, the party who receives s. All parties in $\mathcal{P}_{\texttt{active}}$ compute $[s]_t := [s]_t + [0]_t$ and send their shares of $[s]_t$ to $P_{i^\star}$.
3: Each receiver $P_{i^\star}$ reconstructs s from the shares it receives in the last step.

Let c_O be the number of output gates in the circuit. Then all output gates are divided into $\lceil c_O/T \rceil$ segments. Note that each time $\mathcal{F}_{\text{zero}}$ outputs a pair of disputed parties, at least one corrupted party is removed from $\mathcal{P}_{\texttt{active}}$. Thus $\mathcal{F}_{\text{zero}}$ will be *rerun* at most $O(n)$ times. $\mathcal{F}_{\text{zero}}$ will be invoked at most $O(n) + \lceil c_O/T \rceil$ times. For each output gate, all parties send $O(n)$ elements to the designated party to reconstruct the output. Therefore, by using ZeroShareRandom to instantiate $\mathcal{F}_{\text{zero}}$, the overall communication complexity is $O(c_O n\kappa + n^3\kappa)$ bits.

6.4 Main Protocol

Now, we are ready to present the main protocol. In the protocol, all parties first invoke $\mathcal{F}_{\text{input}}$ to securely share their inputs. Then $\mathcal{F}_{\text{rand}}$ is invoked to generate random sharings for random gates.

Let c_M denote the number of multiplication gates. The circuit is divided into $\lceil c_M/T \rceil$ segments such that each segment contains T multiplication gates. Segments are evaluated sequentially based on their topological order. For each

segment $\mathtt{seg}$, the procedure EVAL is invoked. If the result is a pair of disputed parties, then these two parties are removed from $\mathcal{P}_{\mathtt{active}}$ and all parties in $\mathcal{P}_{\mathtt{active}}$ reevaluate $\mathtt{seg}$. Otherwise, the protocol continues to evaluate the next segment.

Finally, OUTPUT is invoked to reconstruct the outputs to designated parties.

Protocol 18 MAIN

1: Input gates:
   All parties invoke $\mathcal{F}_{\text{input}}(c_I)$.
2: Rand gates:
   All parties invoke $\mathcal{F}_{\text{rand}}(c_R)$.
3: Evaluation:
   1. All parties agree on a partition $(\mathtt{seg}_1, \mathtt{seg}_2, \ldots, \mathtt{seg}_{\lceil c_M/T \rceil})$ of the circuit such that the number of multiplication gates of each segment is T.
   2. From $j = 1$ to $\lceil c_M/T \rceil$:
      (a) All parties run the procedure EVAL$(\mathcal{P}_{\mathtt{active}}, \mathtt{seg}_j)$.
      (b) If the output is $(P_i, P_k, \text{disputed})$, all parties set $\mathcal{P}_{\mathtt{active}} := \mathcal{P}_{\mathtt{active}} \backslash \{P_i, P_k\}$ and repeat the loop.
      (c) Otherwise, set $j := j + 1$ and continue to handle the next segment.
4: Output gates:
   All parties invoke the procedure OUTPUT$(\mathcal{P}_{\mathtt{active}})$.

Now we analyze the communication complexity of MAIN.

For Step 3, each time EVAL outputs a pair of disputed parties, at least one corrupted party is removed from $\mathcal{P}_{\mathtt{active}}$. Thus, EVAL will be *rerun* at most $O(n)$ times. In total, EVAL will be invoked $O(n) + \lceil c_M/T \rceil$ times. The overall communication complexity of this step is $O(c_M n\kappa + n^3)$ bits.

Let $C = c_I + c_R + c_M + c_O$. Then the overall communication complexity of MAIN is $O(Cn\kappa + n^3\kappa)$ bits.

Theorem 1. *Let $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq 2n$ and C be an arithmetic circuit over $\mathbb{F}$. Protocol* MAIN *evaluates C with perfect security against an active adversary which corrupts at most $t < n/3$ parties.*

We provide the full proof of Theorem 1 in Appendix B.

References

ABF+17. Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority mpc for malicious adversariesbreaking the 1 billion-gate per second barrier. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 843–862. IEEE, 2017.

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

BGP92. Piotr Berman, Juan A. Garay, and Kenneth J. Perry. *Bit Optimal Distributed Consensus*, pages 313–321. Springer US, Boston, MA, 1992.

BJMS18. Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. Threshold multi-key fhe and applications to mpc. Cryptology ePrint Archive, Report 2018/580, 2018. https://eprint.iacr.org/2018/580.

BOGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10. ACM, 1988.

BSFO12. Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 663–680, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

BTH08. Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In Ran Canetti, editor, *Theory of Cryptography*, pages 213–230, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

CCD88. David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19. ACM, 1988.

CDVdG87. David Chaum, Ivan B Damgård, and Jeroen Van de Graaf. Multiparty computations ensuring privacy of each partys input and correctness of the result. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 87–119. Springer, 1987.

CGH+18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority mpc for malicious adversaries. In *Annual International Cryptology Conference*, pages 34–64. Springer, 2018.

CW92. Brian A. Coan and Jennifer L. Welch. Modular construction of a byzantine agreement protocol with optimal message bit complexity. *Inf. Comput.*, 97(1):61–85, March 1992.

DI06. Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In *Proceedings of the 26th Annual International Conference on Advances in Cryptology*, CRYPTO'06, pages 501–520, Berlin, Heidelberg, 2006. Springer-Verlag.

DIK10. Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 445–465. Springer, 2010.

DN07. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*, pages 572–590. Springer, 2007.

FLNW17. Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 225–255. Springer, 2017.

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.

HM01. Martin Hirt and Ueli Maurer. Robustness for free in unconditional multiparty computation. In *Annual International Cryptology Conference*, pages 101–118. Springer, 2001.

HMP00. Martin Hirt, Ueli Maurer, and Bartosz Przydatek. Efficient secure multiparty computation. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 143–161. Springer, 2000.

HN06. Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In *Annual International Cryptology Conference*, pages 463–482. Springer, 2006.

LN17. Yehuda Lindell and Ariel Nof. A framework for constructing fast mpc over arithmetic circuits with malicious adversaries and an honest-majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 259–276. ACM, 2017.

LP12. Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of cryptology*, 25(4):680–722, 2012.

LSP82. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology–CRYPTO 2012*, pages 681–700. Springer, 2012.

Pai99. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

RBO89. T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 73–85, New York, NY, USA, 1989. ACM.

Sha79. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

Yao82. Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

A Construction of the Simulator

In this part, we give a description of the simulator $\mathcal{S}$, which will be used to prove the security of Main in the next part.

Suppose $\mathcal{A}$ is the adversary in the real world. We use c_I, c_R, c_M, c_O for the number of input, random, multiplication and output gates respectively. In the beginning, we set $\mathcal{P}_{\texttt{active}}$ to be the set of all parties and $\mathcal{C}_{\texttt{active}}$ to be the set of all corrupted parties.

In Step 1, $\mathcal{S}$ receives the inputs of all corrupted parties. For each input of honest parties, $\mathcal{S}$ sets it to be 0. Then $\mathcal{S}$ faithfully emulates $\mathcal{F}_{\text{input}}$. In this step, $\mathcal{S}$ learns all shares of corrupted parties of all inputs.

In Step 2, $\mathcal{S}$ faithfully emulates $\mathcal{F}_{\text{rand}}$. In this step, $\mathcal{S}$ learns all shares of corrupted parties of all random elements.

In Step 3.1, $\mathcal{S}$ behaves honestly.

In Step 3.2, we describe the behavior of $\mathcal{S}$ when executing Eval.

Initialization: $\mathcal{S}$ behaves honestly.

Generate multiplication tuples: $\mathcal{S}$ invokes the simulator $\mathcal{S}_{\text{multi-tuple}}$ of $\mathcal{F}_{\text{multi-tuple}}$. $\mathcal{S}$ emulates $\mathcal{F}_{\text{multi-tuple}}$. If $\mathcal{S}$ receives $(P_i, P_k, \text{disputed})$, it sets $\mathcal{P}_{\texttt{active}} := \mathcal{P}_{\texttt{active}} \backslash \{P_i, P_k\}, \mathcal{C}_{\texttt{active}} := \mathcal{C}_{\texttt{active}} \backslash \{P_i, P_k\}$ and repeats from Step 3.1. Otherwise, $\mathcal{S}$ learns all shares of corrupted parties of all multiplication tuples.

$\mathcal{S}$ also gets the transcript of $\mathcal{S}_{\text{multi-tuple}}$ during the interaction with $\mathcal{A}$, i.e., all messages $\mathcal{S}_{\text{multi-tuple}}$ received from and sent to $\mathcal{A}$.

Evaluate `seg`: We will maintain the invariance that for each t-sharing of input wires of `seg`, $\mathcal{S}$ learns all shares held by corrupted parties. It holds for the first `seg` since every input wire is either from a random gate or an input gate. Therefore, we only need to guarantee that in the end of a successful evaluation, $\mathcal{S}$ learns all shares held by corrupted parties for each t-sharing of output wires of `seg`. In the following steps, $\mathcal{S}$ will compute the shares of corrupted parties accordingly.

For every addition gate, $\mathcal{S}$ does nothing. For every multiplication gate, each time an honest party needs to send its share of $[d]_{n'-1}$ or $[e]_{n'-1}$ to $P_{\texttt{king}}$, $\mathcal{S}$ sends a random element to $P_{\texttt{king}}$.

For the remaining steps, $\mathcal{S}$ behaves honestly. If $P_{\texttt{king}}$ is an honest party, $\mathcal{S}$ behave honestly by reconstructing the $(n'-1)$-sharings and sending back the results to all other parties.

Check the consistency of $P_{\texttt{king}}$: Since $d^1, \ldots, d^T, e^1, \ldots, e^T$ are public and this step does not involve any private shares, $\mathcal{S}$ behaves honestly in this step.

Recompute all reconstructions: $\mathcal{S}$ first computes the values $d^1, \ldots, d^T, e^1, \ldots, e^T$ should be. Recall that, for every $j \in \{1, \ldots, T\}$, $\mathcal{S}$ learns the shares of $[a^j]_{n'-1}$ and $[b^j]_{n'-1}$ held by corrupted parties. In addition, $\mathcal{S}$ also learns the shares of

$[x^j]_t$ and $[y^j]_t$ held by corrupted parties. Therefore, $\mathcal{S}$ learns all shares of $[d^j]_{n'-1}$ and $[e^j]_{n'-1}$ that they ought to be.

For every $j \in \{1, \ldots, T\}$, based on d^j, e^j and all shares of $[d^j]_t$ and $[e^j]_t$ held by corrupted parties, $\mathcal{S}$ randomly computes the remaining shares as the shares of honest parties.

In the remaining steps, $\mathcal{S}$ behaves honestly.

***Commit randomness used in* GenerateTuples-PE**: $\mathcal{S}$ will use the transcript got from $\mathcal{S}_{\text{multi-tuple}}$ to generate the messages that honest parties should send to corrupted parties.

Recall that in TripleShareRandom, all parties in the real world compute

$$([a^1]_{n'-1}, \ldots, [a^{n'}]_{n'-1}) = M([s^1]_{n'-1}, \ldots, [s^{n'}]_{n'-1}).$$

where M is the hyper-invertible matrix all parties agree on. And each of the last $2t'$ $(n'-1)$-sharings is checked by a different party.

Let $\mathcal{C}_{\texttt{check}}$ be the set of corrupted parties which checks a $(n'-1)$-sharing in TripleShareRandom. Let $t_1 = |\mathcal{C}_{\texttt{active}}|$ and $t_2 = |\mathcal{C}_{\texttt{check}}|$. $\mathcal{S}$ select $n' - T - t_1 - t_2$ honest parties (denoted by $\mathcal{H}'$) and for every $P_j \in \mathcal{H}'$, set the shares of $[s^j]_{n'-1}$ held by honest parties to be uniformly random.

Recall that, for every $P_i \in \mathcal{H}_{\texttt{active}}$ in the real world, it should compute

$$(a_i^1, \ldots, a_i^{n'}) = M(s_i^1, \ldots, s_i^{n'}).$$

Let N be the matrix such that

$$(s_i^j)_{\mathcal{H}_{\texttt{active}} \setminus \mathcal{H}'} = N(a_i^1, \ldots, a_i^T, (a_i^j)_{\mathcal{C}_{\texttt{check}}}, (s_i^j)_{j \in \mathcal{C}_{\texttt{active}} \bigcup \mathcal{H}'})$$

where we use $(s_i^j)_{\mathcal{H}_{\texttt{active}} \setminus \mathcal{H}'}$ for the vector only containing $\{s_i^j\}_{P_j \in \mathcal{H}_{\texttt{active}} \setminus \mathcal{H}'}$ and $(a_i^1, \ldots, a_i^T, (a_i^j)_{\mathcal{C}_{\texttt{check}}}, (s_i^j)_{j \in \mathcal{C}_{\texttt{active}} \bigcup \mathcal{H}'})$ for the vector only containing $\{a_i^1, \ldots, a_i^T\}$, $\{a_i^j\}_{P_j \in \mathcal{C}_{\texttt{check}}}, \{s_i^j\}_{P_j \in \mathcal{C}_{\texttt{active}} \bigcup \mathcal{H}'}$. The existence of N is guaranteed by the property of hyper-invertible matrices.

Note that

$$\begin{aligned}(s_i^j)_{\mathcal{H}_{\texttt{active}} \setminus \mathcal{H}'} &= N(a_i^1, \ldots, a_i^T, (a_i^j)_{\mathcal{C}_{\texttt{check}}}, (s_i^j)_{j \in \mathcal{C}_{\texttt{active}} \bigcup \mathcal{H}'}) \\ &= N(d_i^1, \ldots, d_i^T, (a_i^j)_{\mathcal{C}_{\texttt{check}}}, (s_i^j)_{j \in \mathcal{C}_{\texttt{active}} \bigcup \mathcal{H}'}) - N(x_i^1, \ldots, x_i^T, 0, \ldots, 0)\end{aligned}$$

Let $([U^j]_t)_{P_j \in \mathcal{H}_{\texttt{active}} \setminus \mathcal{H}'} := N([x^1]_t, \ldots, [x^T]_t, [0]_0, \ldots, [0]_0)$. Then $N(x_i^1, \ldots, x_i^T, 0, \ldots, 0)$ is just the vector of the i-th shares of $\{[U^j]_t\}_{P_j \in \mathcal{H}_{\texttt{active}} \setminus \mathcal{H}'}$.

Let N_j be the row of N such that $s_i^j = N_j(a_i^1, \ldots, a_i^T, (a_i^j)_{\mathcal{C}_{\texttt{check}}}, (s_i^j)_{j \in \mathcal{C}_{\texttt{active}} \bigcup \mathcal{H}'})$. Let $V_i^j = N(d_i^1, \ldots, d_i^T, (a_i^j)_{\mathcal{C}_{\texttt{check}}}, (s_i^j)_{j \in \mathcal{C}_{\texttt{active}} \bigcup \mathcal{H}'})$. Therefore, $s_i^j = V_i^j - U_i^j$.

For shares of $[s^j]_{n'-1}$ held by corrupted parties, they have been sent to corrupted parties and thus, are fixed. Therefore, $\{s_i^j\}_{P_i \in \mathcal{C}_{\texttt{active}}}$ are fixed. Also, for $P_i \in \mathcal{C}_{\texttt{active}}$, $\mathcal{S}$ knows the value of U_i^j (which is just a linear combination of the i-th shares of $[x^1]_t, \ldots, [x^T]_t$). Thus, $\mathcal{S}$ sets $V_i^j = s_i^j + U_i^j$.

For $P_j \in \mathcal{H}'$, $\mathcal{S}$ generates $[{}_1s^j]_t, [{}_2s^j]_t, [{}_3s^j]_t$ honestly (since all shares of $[s^j]_{n'-1}$ are generated).

For $P_j \in \mathcal{H}_{\texttt{active}} \backslash \mathcal{H}'$, $\mathcal{S}$ randomly generates $[{}_1V^j]_t, [{}_2V^j]_t, [{}_3V^j]_t$ such that for $k \in \{1,2,3\}$ and $P_\ell \in \mathcal{P}_k$, the ℓ-th share of $[{}_kV^j]_t$ is V^j_ℓ. Finally, set $[{}_ks^j]_t = [{}_kV^j]_t - [U^j]_t$ for $k \in \{1,2,3\}$. Note that $\mathcal{S}$ learns all shares of $[{}_ks^j]_t$ which should be held by corrupted parties.

For $k \in \{1,2,3\}$ and every $P_j \in \mathcal{H}_{\texttt{active}}$, $\mathcal{S}$ sends the ℓ-th shares of $[{}_ks^j]_t$ to P_ℓ for all $P_\ell \in \mathcal{C}_{\texttt{active}} \backslash \mathcal{P}_k$.

Check 4-Consistency:

In Check4Consistency, $\mathcal{S}$ first faithfully emulates $\mathcal{F}_{\text{4-consistency}}$ to generate random 4-consistent tuples. If $\mathcal{S}$ receives $(P_i, P_k, \text{disputed})$, it sets $\mathcal{P}_{\texttt{active}} := \mathcal{P}_{\texttt{active}} \backslash \{P_i, P_k\}, \mathcal{C}_{\texttt{active}} := \mathcal{C}_{\texttt{active}} \backslash \{P_i, P_k\}$ and repeats from Step 3.1. Otherwise, $\mathcal{S}$ receives all shares of corrupted parties of all random 4-consistent tuples.

When all shares of a random 4-consistent tuple are sent to a corrupted party, $\mathcal{S}$ honestly generates this tuple and sends the shares to the corrupted party on behalf of honest parties.

For each honest party P_j, $\mathcal{S}$ generates a random 4-consistent tuple $[\![u^j]\!]$ such that for every corrupted party P_ℓ, $({}_0u^j_\ell, {}_1u^j_\ell, {}_2u^j_\ell, {}_3u^j_\ell) = ({}_0s^j_\ell, {}_1s^j_\ell, {}_2s^j_\ell, {}_3s^j_\ell) + ({}_0r^j_\ell, {}_1r^j_\ell, {}_2r^j_\ell, {}_3r^j_\ell)$.

$\mathcal{S}$ behaves honestly in the remaining steps of Check4Consistency.

Find a disputed pair of parties:

For $j \in \{1,2,3\}$, $\mathcal{S}$ randomly generates $[{}_jd^{i^\star}]_t$ such that for $P_k \in \mathcal{P}_j \backslash \mathcal{C}_{\texttt{active}}$, ${}_jd^{i^\star}_k$ is the k-th share of $[d^{i^\star}]_{n'-1}$, which is determined before, and for $P_k \in \mathcal{C}_{\texttt{active}}$, ${}_jd^{i^\star}_k = x^{i^\star}_k + M_{i^\star}({}_js^1_k, \ldots, {}_js^{n'}_k)$.Then $\mathcal{S}$ behaves honestly in the remaining steps.

So far, we have shown the behavior of $\mathcal{S}$ when executing Eval. $\mathcal{S}$ behaves honestly in the remaining sub-steps of Step 3.2.

In Step 4, $\mathcal{S}$ first faithfully emulates $\mathcal{F}_{\text{zero}}$ to generate random t-sharings of 0. If the output is $(P_i, P_k, \text{disputed})$, $\mathcal{S}$ sets $\mathcal{P}_{\texttt{active}} := \mathcal{P}_{\texttt{active}} \backslash \{P_i, P_k\}, \mathcal{C}_{\texttt{active}} := \mathcal{C}_{\texttt{active}} \backslash \{P_i, P_k\}$ and redo the loop. Otherwise, $\mathcal{S}$ receives all shares of the random t-sharings of 0 held by corrupted parties.

$\mathcal{S}$ then invokes the trusted party/ideal functionality with the inputs of corrupted parties received in Step 1 and receives the outputs of corrupted parties.

For every output gate to an honest party, $\mathcal{S}$ does nothing.

For every output gate to a corrupted party, let $\{r_i\}_{P_i \in \mathcal{C}_{\texttt{active}}}$ be the elements $\mathcal{A}$ provides for a random t-sharing of 0 associated with this gate. Let v be the output of this gate $\mathcal{S}$ received from the trusted party/ideal functionality. Note that $\mathcal{S}$ learns the shares of $[v]_t$ held by corrupted parties. $\mathcal{S}$ randomly generates a new t-sharing of v, namely $[v']_t$, such that for $P_i \in \mathcal{C}_{\texttt{active}}$, $v'_i = v_i + r_i$. $\mathcal{S}$ sends the shares of $[v']_t$ to the designated corrupted party on behalf of honest parties.

B Proof of the Security

In this part, we prove Theorem 1. Formally,

Theorem 1. *Let $\mathbb{F}$ be a finite field of size $|\mathbb{F}| \geq 2n$ and C be an arithmetic circuit over $\mathbb{F}$. Protocol* Main *evaluates C with perfect security against an active adversary which corrupts at most $t < n/3$ parties.*

Proof. We only need to show that, the view of $\mathcal{A}$ when interacting with the simulator $\mathcal{S}$ we constructed in the last part has the same distribution as that in the real world. Consider the following hybrids.

Hybrid$_0$: Execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ emulate all functionalities in the protocol faithfully. Specifically, $\mathcal{S}$ emulate $\mathcal{F}_{\text{input}}$ in Step 1 with real inputs of honest parties, $\mathcal{F}_{\text{rand}}$ in Step 2, $\mathcal{F}_{\text{4-consistency}}$ in Step 3 and $\mathcal{F}_{\text{zero}}$ in Step 4. The distribution is the same as **Hybrid$_0$**.

Hybrid$_2$: In this hybrid, $\mathcal{S}$ invokes $\mathcal{S}_{\text{multi-tuple}}$ and emulates $\mathcal{F}_{\text{multi-tuple}}$ instead of running GenerateTuple-PE in Step 3. The distribution is the same as **Hybrid$_1$**.

Hybrid$_3$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\text{zero}}$ and simulates the behaviors of honest parties in Step 4 by using the strategy of Step 4. Note that $\mathcal{S}$ receives the inputs of corrupted parties when emulating $\mathcal{F}_{\text{input}}$.

The distribution is identical with **Hybrid$_2$**.

Hybrid$_4$: In this hybrid, $\mathcal{S}$ generates $[{}_1d^{i^\star}]_t, [{}_2d^{i^\star}]_t, [{}_3d^{i^\star}]_t$ by using the strategy of finding a disputed pair of parties. We claim that the distribution is identical with **Hybrid$_3$**.

To see this, for every $j \in \{1,2,3\}$ and $P_k \in \mathcal{C}_{\texttt{active}}$, the k-th share of $[{}_jd^{i^\star}]_t$ is fixed in either hybrid. And for $P_k \in \mathcal{P}_j \backslash \mathcal{C}_{\texttt{active}}$, the k-th share of $[{}_jd^{i^\star}]_t$ is just the k-th share of $[d^{i^\star}]_{n'-1}$, which is also fixed in either hybrid.

Thus if the size of $|\mathcal{P}_j \bigcup \mathcal{C}_{\texttt{active}}| \geq t+1$, then all shares of $[{}_jd^{i^\star}]_t$ are determined. Otherwise, recall that

$$[{}_jd^{i^\star}]_t = [x^{i^\star}]_t + M_{i^\star}([{}_js^1]_t, \ldots, [{}_js^{n'}]_t)$$

For each honest party P_k, $[{}_js^k]_t$ is a random t-sharing such that for every $P_i \in \mathcal{P}_j$, ${}_js_i^k$ is the i-th share of $[s^k]_{n'-1}$. For $P_\ell \in \mathcal{C}_{\texttt{active}}$, ${}_js_\ell^k$ is fixed since it has been sent to a corrupted party. Therefore, $[{}_js^k]_t$ is a random t-sharing after fixing the shares held by $\mathcal{P}_j \bigcup \mathcal{C}_{\texttt{active}}$. Thus, $[{}_jd^{i^\star}]_t$ is a random t-sharing after fixing the shares held by $\mathcal{P}_j \bigcup \mathcal{C}_{\texttt{active}}$, whose distribution is exactly the same as that generated by $\mathcal{S}$.

Hybrid$_5$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\text{4-consistency}}$ and use the strategy of checking 4-consistency to generate $[\![u^j]\!]$ for each honest party P_j.

The distribution is the same as **Hybrid$_4$**.

Hybrid$_6$: In this hybrid, $\mathcal{S}$ generates $[{}_1s^i]_t, [{}_2s^i]_t, [{}_3s^i]_t$ for each honest party P_i by using the strategy of committing randomness used in GenerateTuples-PE. We claim the distribution remains the same.

To see this, we first analyze what requirements $[s^1]_{n'-1}, \ldots, [s^{n'}]_{n'-1}$ should satisfy. Note that, before this step, they were only used when generating random triple-sharings.

For every honest party P_j (as a dealer), shares of $[s^j]_{n'-1}$ held by corrupted parties are fixed since they have been sent to corrupted parties. Therefore, we only focus on the shares held by honest parties.

In TripleShareRandom, all parties compute

$$([a^1]_{n'-1}, \ldots, [a^{n'}]_{n'-1}) = M([s^1]_{n'-1}, \ldots, [s^{n'}]_{n'-1}).$$

where M is the hyper-invertible matrix all parties agree on.

Therefore, for every honest party P_i (as a receiver), the shares held by P_i should satisfy

$$(a_i^1, \ldots, a_i^{n'}) = M(s_i^1, \ldots, s_i^{n'})$$

Note that for $j \in \{1, \ldots, T\}$, a_i^j is fixed and $a_i^j = d_i^j - x_i^j$ where d_i^j is the i-th share of $[d^j]_{n'-1}$ and x_i^j is the i-th share of $[x^j]_t$. Also for $P_j \in \mathcal{C}_{\texttt{active}}$, s_i^j is fixed since it was received from a corrupted party. In addition, for $P_j \in \mathcal{C}_{\texttt{check}}$ which checks $[a^j]_{n'-1}$ in TripleShareRandom (i.e., $j \geq T+1$), a_i^j is fixed since it has been sent to a corrupted party. In total, $(T + t_1 + t_2)$ $(\leq n')$ of $a_i^1, \ldots, a_i^{n'}, s_i^1, \ldots, s_i^{n'}$ are fixed. The distribution of $\{s_i^j\}_{P_j \in \mathcal{H}_{\texttt{active}}}$ is random while satisfying the constrain

$$(a_i^1, \ldots, a_i^{n'}) = M(s_i^1, \ldots, s_i^{n'})$$

By the property of hyper invertible matrices, any n' values of $a_i^1, \ldots, a_i^{n'}, s_i^1, \ldots, s_i^{n'}$ can determine the remaining n' values. Therefore, we can view $(n' - T - t_1 - t_2)$ elements of $\{s_i^j\}_{P_j \in \mathcal{H}}$ are uniformly random and the remaining are then determined.

In this hybrid, $\mathcal{S}$ chooses a set of $(n' - T - t_1 - t_2)$ honest parties (denoted by $\mathcal{H}'$) and set s_i^j to be uniformly random for all $P_j \in \mathcal{H}'$. That is, $\mathcal{S}$ simulates the parties in $\mathcal{H}'$ honestly by generating all shares of $[s^j]_{n'-1}$ for all $P_j \in \mathcal{H}'$. (Recall that, for $P_j \in \mathcal{H}_{\texttt{active}} \backslash \mathcal{H}'$, $\mathcal{S}$ only generates the shares of $[s^j]_{n'-1}$ held by corrupted parties). Now $\{s_i^j\}_{j \in \mathcal{H}_{\texttt{active}} \backslash \mathcal{H}'}$ are determined by $\{a_i^1, \ldots, a_i^T\}, \{a_i^j\}_{P_j \in \mathcal{C}_{\texttt{check}}}, \{s_i^j\}_{P_j \in \mathcal{C}_{\texttt{active}} \bigcup \mathcal{H}'}$. And the distribution of $\{[s^j]_{n'-1}\}_{P_j \in \mathcal{H}_{\texttt{active}}}$ is identical with the real one.

$\mathcal{S}$ uses $\{[s^j]_{n'-1}\}_{P_j \in \mathcal{H}_{\texttt{active}}}$ it generated (instead of the real one) to generate $\{[_1 s^j]_t, [_2 s^j]_t, [_3 s^j]_t\}$ faithfully. Thus the distribution is the same as $\mathbf{Hybrid_5}$.

We point out that, in the generating process, $\mathcal{S}$ does not directly use the specific values of $\{a_i^1, \ldots, a_i^T\}_{P_i \in \mathcal{H}_{\texttt{active}}}$. Instead, $\mathcal{S}$ only uses the values of $\{d_i^1, \ldots, d_i^T\}_{P_i \in \mathcal{H}_{\texttt{active}}}$ and all shares of $[x^1]_t, \ldots, [x^T]_t$ held by corrupted parties.

$\mathbf{Hybrid_7}$: In this hybrid, $\mathcal{S}$ computes the shares of $[d^1]_t, \ldots, [d^T]_t, [e^1]_t, \ldots, [e^T]_t$ held by honest parties, which are used to recompute all reconstructions in Step 3, instead of using the real one by using the strategy of recomputing all reconstructions.

Although x^j or y^j may be shared to a large set of parties (for example, x^j is an input and therefore even parties which are not in $\mathcal{P}_{\texttt{active}}$ get shares), a^j and

b^j are only shared to $\mathcal{P}_{\texttt{active}}$. Thus $[d^j]_t = [x^j]_t + [a^j]_t$ and $[e^j]_t = [y^j]_t + [b^j]_t$ are uniformly distributed such that the shares held by corrupted parties are fixed.

Therefore, the distribution is the same as $\mathbf{Hybrid_6}$.

$\mathbf{Hybrid_8}$: In this hybrid, $\mathcal{S}$ uses the strategy of evaluating `seg` when evaluating every multiplication gate and addition gate. Explicitly, each time an honest party needs to send its share to $P_{\texttt{king}}$, $\mathcal{S}$ sends a uniform element to $P_{\texttt{king}}$ instead of the real share.

Note that the shares of $[a]_{n'-1}$ held by honest parties are uniformly distributed. To see this, consider the process that $\mathcal{F}_{\text{multi-tuple}}$ generates a multiplication tuple. On receiving the shares held by corrupted parties, $\mathcal{F}_{\text{multi-tuple}}$ first sets random elements to the shares of $[a]_{n'-1}$ held by honest parties. Combining with the shares provided by $\mathcal{S}_{\text{multi-tuple}}$, a is determined. Then $[a]_t$ is generated. $[b]_{t,n'-1}$ are generated in a similar way. After that, $c = ab$ is determined and $\mathcal{F}_{\text{multi-tuple}}$ generates $[c]_t$. Therefore, the shares of $[d]_{n'-1} = [x]_t + [a]_{n'-1}$ and $[e]_{n'-1} = [y]_t + [b]_{n'-1}$ held by honest parties are uniformly distributed. The distribution is the same as $\mathbf{Hybrid_7}$.

$\mathbf{Hybrid_9}$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\text{multi-tuple}}$ by using the strategy of generating multiplication tuples. Note that, the only difference between this hybrid and $\mathbf{Hybrid_8}$ is that the shares of all multiplication tuples held by honest parties are not generated. However, in $\mathbf{Hybrid_8}$, these shares are not used after they are generated. Thus, the distribution is the same as $\mathbf{Hybrid_8}$.

$\mathbf{Hybrid_{10}}$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ by using the strategy of Step 2. Note that, the only difference between this hybrid and $\mathbf{Hybrid_9}$ is that the shares of all random t-sharings held by honest parties are not generated. However, in $\mathbf{Hybrid_9}$, these shares are not used after they are generated. Thus, the distribution is the same as $\mathbf{Hybrid_9}$.

$\mathbf{Hybrid_{11}}$: In this hybrid, $\mathcal{S}$ emulates $\mathcal{F}_{\text{input}}$ by using the strategy of Step 1. The only difference between this hybrid and $\mathbf{Hybrid_{10}}$ is that $\mathcal{S}$ provides 0 for each input of honest parties instead of the real one. However, it only changes the shares held by honest parties since the shares held by corrupted parties are provided by $\mathcal{A}$. And these shares are not used after they are generated. Thus, the distribution is the same as $\mathbf{Hybrid_{10}}$.

Note that $\mathbf{Hybrid_{11}}$ is the execution between $\mathcal{S}$ and $\mathcal{A}$ in the ideal world.

We conclude that the distribution of $\mathbf{Hybrid_{11}}$ is the same as $\mathbf{Hybrid_0}$. □