Malicious Security Comes Free in Honest-Majority MPC

Vipul Goyal and Yifan Song

Carnegie Mellon University, Pittsburgh, USA
vipul@cmu.edu, yifans2@andrew.cmu.edu

Abstract. We study the communication complexity of unconditionally secure MPC over point-to-point channels for corruption threshold $t < n/2$. We ask the question: "is it possible to achieve security-with-abort with the same concrete cost as the best-known semi-honest MPC protocol?" While a number of works have focused on improving the concrete efficiency in this setting, the answer to the above question has remained elusive until now.

We resolve the above question in the affirmative by providing a secure-with-abort MPC protocol with the same cost per gate as the best-known semi-honest protocol. Concretely, our protocol only needs 5.5 field elements per multiplication gate per party which matches (and even improves upon) the corresponding cost of the best known protocol in the semi-honest setting by Damgård and Nielsen. Previously best-known maliciously secure (with abort) protocols require 12 field elements. An additional feature of our protocol is its conceptual simplicity. Our result is achieved by relying on a novel technique of verifying a batch of multiplication tuples.

1 Introduction

In secure multiparty computation (MPC), a set of n parties together evaluate a function f on their private inputs. This function f is public to all parties, and, may be modeled as an arithmetic circuit over a finite field. Very informally, a protocol of secure multiparty computation guarantees the privacy of the inputs of every (honest) individual except the information which can be deduced from the output. This notion was first introduced in the work [Yao82] of Yao. Since the early feasibility solutions proposed in [Yao82,GMW87], various settings of MPC have been studied. Examples include semi-honest security vs malicious security, security against computational adversaries vs unbounded adversaries, honest majority vs corruptions up to $n-1$ parties, security with abort vs guaranteed output delivery and so on.

In this work, we focus on the information-theoretical setting (i.e., security against unbounded adversaries). The adversary is allowed to corrupt at most $t < n/2$ parties and is fully malicious. We assume the existence of a private point-to-point communication channel between every pair of parties. We are interested in the communication complexity of the secure MPC, which is measured by the

number of bits via private point-to-point channels. To achieve the best efficiency, our protocol allows a premature abort in the computation (i.e., security-with-abort) and does not achieve fairness or guaranteed output delivery.

The first positive solutions in this setting were proposed in [RBO89,Bea89] and the focus subsequently shifted to efficiency. In particular, several recent works [GIP+14a,LN17,CGH+18,NV18] have focused on improving the communication complexity. Genkin et al. [GIP+14a] provided the first construction with malicious security (with abort) having the same asymptotic communication complexity as the best-known semi-honest protocol [DN07] (hereafter referred to as the DN protocol). Since then, the main focus in this line of research has been to improve the concrete communication complexity per gate. Compared with the DN protocol, the recent breakthrough [CGH+18,NV18] showed that achieving security-with-abort requires only twice the cost of achieving semi-honest security. In the setting of 1/3 corruption threshold, a recent beautiful work of Furukawa and Lindell [FL19] presented a construction which achieves the same communication cost as the DN protocol. When considering a 3-party computation for a binary circuit, a recent work [ABF+17] presented a construction where each AND gate only requires 7 bits per party. As a result, over a billion AND gates could be processed within one second.

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

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

The best-known results in this setting [CGH+18,NV18] achieved concrete efficiency of 12 field elements per multiplication gate, while the best-known semi-honest result [DN07] only requires 6 field elements per multiplication gate. Note that, by representing the functionality as an arithmetic circuit, the communication complexity of the protocol in the unconditional setting is typically dominated by the number of multiplication gates in the circuit. This is because the addition gates can usually be done locally, requiring no communication at all.

1.1 Our Results.

In this work, we answer the above question in the affirmative by presenting an MPC protocol with concrete efficiency 5.5 field elements per gate, which matches (and even improves upon) the concrete cost of the best-known result [DN07] in the semi-honest setting. Our result is achieved by using a novel technique to efficiently verify a batch of multiplications. We observe that the additional cost in [CGH+18,NV18] comes from the verification of the multiplications. Our new technique brings down the cost to a term that only has a sub-linear dependence on the circuit size. In this way, the cost of the verification no longer affects the concrete efficiency, and our result achieves the same concrete efficiency as the DN protocol. Our protocol additionally makes a simple optimization to the DN

protocol, which brings down the cost from 6 field elements per gate to 5.5 field elements per gate. A sketch of our new technique can be found in Section 2.

A particularly attractive feature of our protocol is its relative simplicity. Compared with the constructions in [CGH$^+$18,NV18], we also remove several checks to make the protocol as succinct as possible. Specifically, our new technique of verifying a batch of multiplication tuples is the only check in the protocol and the remaining parts are the same as the semi-honest DN protocol. In particular, we do not check the consistency/validity of the sharings as required in [CGH$^+$18,NV18].

Furthermore, the security of our construction does not depend upon the field size. One can use a field with size as low as $n + 1$ where n is the number of parties. On the other hand, the concrete efficiency of both constructions from [CGH$^+$18,NV18] suffers from having a large field size. An alternative presented in [CGH$^+$18] is to use a small field but then the verification must be done several times to reach the desired security parameter. This however would increase the number of field elements per multiplication gate several times. Another option presented in [NV18] allows one to reduce the field size without substantially increasing the number of fields elements per gate. However, the field size must still be at least as large as the circuit size and also depends upon the security parameter (and, e.g., cannot be a constant).

1.2 Related Works

In this section, we compare our result with several related constructions in both techniques and the efficiency. In the following, let C denote the size of the circuit, ϕ denote the size of a field element, κ denote the security parameter, and n denote the number of parties participating in the computation.

Comparison with [DN07,GIP$^+$14a]. In [DN07], Damgård and Nielsen introduced the best-known semi-honest protocol, which we refer to as the DN protocol. The communication complexity of the DN protocol is $O(Cn\phi + n^2\phi)$ bits. The concrete efficiency is 6 field elements per gate (per party).

In [GIP$^+$14a], Genkin, et al. showed that the DN protocol is secure up to an additive attack when running in the fully malicious setting. Based on this observation, a secure-with-abort MPC protocol can be constructed by combining the DN protocol and a circuit which is resilient to an additive attack (referred to as an AMD circuit). As a result, [GIP$^+$14a] gave the first construction against a fully malicious adversary with communication complexity $O(Cn\phi+n^2\phi)$ bits (for a large enough field), which matches the asymptotic communication complexity of the DN protocol.

Our construction relies on another observation that the DN protocol provides perfect privacy of honest parties (before the output phase) in the presence of a fully malicious adversary. This observation can be seen as a corollary of the theorem showed in [GIP$^+$14a], and is also used in several other works [LN17,NV18]. In this way, the only task is to check the correctness of the computation before

the output phase. As a result, we do not need to compile a circuit into an AMD circuit and therefore save the cost introduced by the compilation.

As for the concrete efficiency, our result achieves the same cost per gate as the DN protocol, which means that security-with-abort can be obtained with *no additional communication cost*.

Comparison with [CGH+18]. The construction in [CGH+18] also relied on the theorem showed in [GIP+14a]. The idea is to check whether the adversary launched an additive attack. In the beginning, all parties compute a random secret sharing of the value r. For each wire w with the value x associated with it, all parties will compute two secret sharings of the secret values x and $r \cdot x$ respectively. Here $r \cdot x$ can be seen as a secure MAC of x when the only possible attack is an additive attack. In this way, the protocol requires two operations per multiplication gate. The asymptotic communication complexity is $O(Cn\phi+n^2\phi)$ bits (for a large enough field) and the concrete efficiency is 12 field elements per gate.

Our construction relies on a different observation and achieves 5.5 field elements per gate. Also, our construction does not require the field size to be large enough and works for a field of size $n+1$ (which is necessary for the underlying secret sharing scheme with n parties).

Comparison with [LN17,NV18]. These two constructions are based on the same observation that the DN protocol provides perfect privacy of honest parties in the presence of a fully malicious adversary.

In [NV18], Nordholt and Veeningen used the Batch-wise Multiplication Verification technique introduced in [BSFO12] to check the correctness of all multiplication gates, which is sufficient since these are the only operations that require interaction. This technique needs one more multiplication operation per multiplication gate. Therefore, the construction achieves the asymptotic communication complexity $O(Cn\phi + n^2\phi)$ bits (for a large enough field) and the concrete efficiency 12 field elements per gate.

Technically, we extend the Batch-wise Multiplication Verification technique so that it no longer requires an additional multiplication operation per multiplication gate. As a result, we reduce the cost of the check from 6 field elements per gate to $o(1)$ field elements per gate. Also our construction does not require the field size to be large enough.

Comparison with [FL19]. In the setting of 1/3 corruption threshold, Furukawa and Lindell [FL19] achieved the same communication complexity as the best semi-honest protocol [DN07]. Concretely, the communication cost per gate is $4\frac{2}{3}$ elements per party. Technically, multiplications can be verified in a much simpler way than that in the setting of honest majority. This is because a degree-$2t$ Shamir secret sharing can be fully determined by the shares of honest parties since there are at least $2t+1$ honest parties, where t is the corruption threshold.

In more detail, suppose that we want to check the correctness of a multiplication tuple $([x]_t, [y]_t, [x \cdot y]_t)$ where $[a]_t$ refers to a degree-t Shamir secret sharing

of the secret value a. By the property of the Shamir secret sharing, all parties can locally compute a degree-$2t$ sharing of the product, i.e., $[x \cdot y]_{2t}$. Therefore, the verification becomes the check of whether $[x \cdot y]_{2t}$ and $[x \cdot y]_t$ have the same secret value.

However, this way of verification completely fails in the setting of honest majority since a degree-$2t$ sharing cannot be determined by the shares of honest parties. In particular, an adversary can simply modify the secret value by changing only one share. On the other hand, our technique is more general and can also be used in the setting of $1/3$ corruption threshold to achieve the same communication cost per gate as [FL19] (though it will increase the cost that is independent of the circuit).

Other Related Works. The notion of MPC was first introduced in [Yao82] and [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.

A series of works 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 of honest majority, a public broadcast channel is required. A rich line of works [CDD+99,BTH06,BSFO12,IKP+16] have focused on improving the asymptotic communication complexity in this setting. In the setting of $1/3$ corruption threshold, a public broadcast channel can be securely simulated and therefore, only private point-to-point communication channels are required. A rich line of works [HMP00,HM01,DN07,BTH08,GLS19] have focused on improving the asymptotic communication complexity in this setting. In the setting where $t < (1/3 - \epsilon)n$, packed secret sharing can be used to hide a batch of values, resulting in more efficient protocols. E.g., Damgård 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.

When the number of corrupted parties is bounded by $(1/2 - \epsilon)n$, Genkin et al. [GIP+14a] showed that one can even achieve sub-constant cost per gate relying on packed secret sharing. Several works have also focused on the performance of 2-party computation and 3-party computation in practice. Examples include [LP12,NNOB12] for 2-party computation, [FLNW17,ABF+17] for 3-party computation and so on. All of these works emphasized on the practical running time and provided security with abort.

2 Technical Overview

In this section, we give a high-level overview of our technique.

2.1 Notations

In the following, we will use n to denote the number of parties and t to denote the number of corrupted parties. In the setting of the honest majority, we have $n = 2t + 1$.

The construction is based on Shamir Secret Sharing Scheme [Sha79]. We will use $[x]_d$ to denote a degree-d sharing, or a $(d+1)$-out-of-n Shamir sharing. It requires at least $d + 1$ shares to reconstruct the secret and any d shares do not leak any information about the secret.

2.2 General Strategy and Protocol Overview

In [GIP+14b], Genkin, et al. showed that several semi-honest MPC protocols are secure up to an additive attack in the presence of a fully malicious adversary. As one corollary, these semi-honest protocols provide full privacy of honest parties before reconstructing the output. Therefore, a straightforward strategy to achieve security-with-abort is to (1) run a semi-honest protocol till the output phase, (2) check the correctness of the computation, and (3) reconstruct the output only if the check passes.

Fortunately, the best-known semi-honest protocol in this setting [DN07] is secure up to an additive attack. Our construction will follow the above strategy. The main task is the second step, i.e., checking the correctness of the computation before reconstructing the final results.

2.3 Review: DN Semi-Honest Protocol

The best-known semi-honest protocol was proposed in the work of Damgård and Nielsen [DN07]. The protocol consists of 4 phases: Preparation Phase, Input Phase, Computation Phase, and Output Phase. Here we give a brief description of these four phases.

Preparation Phase. In the preparation phase, all parties need to prepare several random sharings which will be used in the computation phase. Specifically, there are two kinds of random sharings needed to be prepared. The first kind is a random degree-t sharing $[r]_t$. The second kind is a pair of random sharings $([r]_t, [r]_{2t})$, which is referred to as double sharings. At a high-level, these two kinds of random sharings are prepared in the following manner:

1. Each party generates and distributes a random degree-t sharing (or a pair of random double sharings).
2. Each random sharing (or each pair of double sharings) is a linear combination of the random sharings (or the random double sharings) distributed by each party.

More details can be found in Section 3.3 and Section 3.4.

Input Phase. In the input phase, each input holder generates and distributes a random degree-t sharing of its input.

Computation Phase. In the computation phase, all parties need to evaluate addition gates and multiplication gates. For an addition gate with input sharings $[x]_t, [y]_t$, all parties just locally add their shares to get $[x+y]_t = [x]_t + [y]_t$. For a multiplication gate with input sharings $[x]_t, [y]_t$, one pair of double sharings $([r]_t, [r]_{2t})$ is consumed. All parties execute the following steps.

1. All parties first locally compute $[x \cdot y + r]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[x \cdot y + r]_{2t}$ and reconstructs the value $x \cdot y + r$. Then $P_{\texttt{king}}$ sends the value $x \cdot y + r$ back to all other parties.
3. All parties locally compute $[x \cdot y]_t = x \cdot y + r - [r]_t$.

Here $P_{\texttt{king}}$ is the party all parties agree on in the beginning.

Output Phase. In the output phase, all parties send their shares of the output sharing to the party who should receive this result. Then that party can reconstruct the output.

Improvement to 5.5 Field Elements. We note that in the second step of the multiplication protocol, $P_{\texttt{king}}$ can alternatively generate a degree-t sharing $[x \cdot y + r]_t$ and distribute the sharing to all other parties. Then in the third step, $[x \cdot y]_t$ can be computed by $[x \cdot y + r]_t - [r]_t$. In fact, $P_{\texttt{king}}$ can set the shares of (a predetermined set of) t parties to be 0 in the sharing $[x \cdot y + r]_t$. This means that $P_{\texttt{king}}$ need not to communicate these shares at all, reducing the communication by half. We rely on the following two observations:

- While normally setting some shares to be 0 could compromise the privacy of the secret (by effectively reducing the reconstruction threshold), note that here $x \cdot y + r$ need not to be private at all.
- Parties do not actually need to receive $x \cdot y + r$ from $P_{\texttt{king}}$. Rather, receiving shares of $x \cdot y + r$ is sufficient to allow them to proceed in the protocol.

This simple observation leads to an improvement of reducing the cost per gate from 6 elements to 5.5 elements. Note that in this construction, all multiplication gates at the same "layer" in the circuit can be evaluated in parallel. Hence, it is even possible to perform a "load balancing" such that the overall cost of different parties roughly remains the same.

2.4 Review: Batch-wise Multiplication Verification

This technique is introduced in the work of Ben-Sasson, et al. [BSFO12]. It is used to check a batch of multiplication tuples efficiently. Specifically, given m multiplication tuples

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

we want to check whether $x^{(i)} \cdot y^{(i)} = z^{(i)}$ for all $i \in [m]$.

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

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

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

All parties first locally compute $[f(\cdot)]_t$ and $[g(\cdot)]_t$ using $\{[x^{(i)}]_t\}_{i\in[m]}$ and $\{[y^{(i)}]_t\}_{i\in[m]}$ respectively. Here a degree-t sharing of a polynomial means that each coefficient is secret-shared. For $i \in \{m+1, \ldots, 2m-1\}$, all parties locally compute $[f(i)]_t, [g(i)]_t$ and then compute $[z^{(i)}]_t$ using the multiplication protocol in [DN07]. Finally, all parties locally compute $[h(\cdot)]_t$ using $\{[z^{(i)}]_t\}_{i\in[2m-1]}$.

Note that if $x^{(i)} \cdot y^{(i)} = z^{(i)}$ for all $i \in [2m-1]$, then we have $f \cdot g = h$. Otherwise, we must have $f \cdot g \neq h$. Therefore, it is sufficient to check whether $f \cdot g = h$. Since $h(\cdot)$ is a degree-$2(m-1)$ polynomials, in the case that $f \cdot g = h$, the number of x such that $f(x) \cdot g(x) = h(x)$ holds is at most $2(m-1)$. Thus, it is sufficient to test whether $f(x) \cdot g(x) = h(x)$ for a random x. As a result, this technique compresses m checks of multiplication tuples to a single check of the tuple $([f(x)]_t, [g(x)]_t, [h(x)]_t)$. A secure technique for checking the tuple $([f(x)]_t, [g(x)]_t, [h(x)]_t)$ was given in [BSFO12,NV18].

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

2.5 Extensions

We would like to introduce two natural extensions of the DN multiplication protocol and the Batch-wise Multiplication Verification technique respectively.

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

Specifically, let $\odot$ denote the inner-product operation. Given two input vectors of sharings $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t$, we can compute $[\boldsymbol{x} \odot \boldsymbol{y}]_t$ using the same strategy as the DN multiplication protocol and in particular, with the *same communication cost*. This is because, just like in the multiplication protocol, here all the parties can *locally* compute the shares of the result. These shares are then randomized and sent to $P_{\texttt{king}}$ for degree reduction. More details can be found in Section 4.1. This extension is observed in [CGH+18].

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

Specifically, given a set of m inner-product tuples $\{([\boldsymbol{x}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t, [z^{(i)}]_t)\}_{i\in[m]}$, we want to check whether $\boldsymbol{x}^{(i)} \odot \boldsymbol{y}^{(i)} = z^{(i)}$ for all $i \in [m]$. Here $\{\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\}_{i\in[m]}$ are vectors of the same dimension. The only difference is that all parties will compute $\boldsymbol{f}(\cdot), \boldsymbol{g}(\cdot)$ such that

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

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

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

Then, it is sufficient to test whether $\boldsymbol{f}(x) \odot \boldsymbol{g}(x) = h(x)$ for a random x. As a result, this technique compresses m checks of inner-product tuples to a single check of the tuple $([\boldsymbol{f}(x)]_t, [\boldsymbol{g}(x)]_t, [h(x)]_t)$. It is worth noting that the communication cost remains the *same* as the original technique. More details can be found in 4.2. This extension is observed in [NV18].

Using these Extensions for Reducing the Field Size. We point out that these extensions are not used in any way in the main results of [CGH$^+$18,NV18]. In [CGH$^+$18], the primary purpose of the extension is to check more efficiently in a small field. In more detail, [CGH$^+$18] has a "secure MAC" associated with each wire value in the circuit. At a later point, the MACs are verified by computing a linear combination of the value-MAC pairs with random coefficients. Unlike the case in a large field, the random coefficients cannot be made public due to security reasons. Then a computation of a linear combination becomes a computation of an inner-product. [CGH$^+$18] relies on the extension of the DN multiplication protocol to efficiently compute the inner-product of two vector of sharings. However we note that with the decrease in the field size, the number of field elements required per gate grows up and hence the concrete efficiency goes down. In [NV18], the extension of the Batch-wise Multiplication Verification technique is only pointed out as a corollary of independent interest.

2.6 Fast Verification for a Batch of Multiplication Tuples

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

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

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

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

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

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

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

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

All parties will generate a random degree-t sharing $[r]_t$ in the same way as that in the preparation phase of the DN protocol. Then they reconstruct the value r. We can set

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

Then $F(r) = \boldsymbol{x} \odot \boldsymbol{y}$ and $G(r) = z$. The inner-product tuple $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$ is what we wish to verify.

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

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

To utilize the extension, let $\ell = m/k$ and

$$[\boldsymbol{x}]_t = ([\boldsymbol{a}^{(1)}]_t, [\boldsymbol{a}^{(2)}]_t, \ldots, [\boldsymbol{a}^{(k)}]_t)$$
$$[\boldsymbol{y}]_t = ([\boldsymbol{b}^{(1)}]_t, [\boldsymbol{b}^{(2)}]_t, \ldots, [\boldsymbol{b}^{(k)}]_t),$$

where $\{\boldsymbol{a}^{(i)}, \boldsymbol{b}^{(i)}\}_{i\in[k]}$ are vectors of dimension ℓ. For each $i \in [k-1]$, we compute $[c^{(i)}]_t = [\boldsymbol{a}^{(i)} \odot \boldsymbol{b}^{(i)}]_t$ using the extension of the DN multiplication protocol. Then set $[c^{(k)}]_t = [z]_t - \sum_{i=1}^{k-1} [c^{(i)}]_t$. In this way, if the original tuple is incorrect, then at least one of the new inner-product tuples is incorrect.

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

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

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

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

The random multiplication tuple is prepared in the following manner.

1. All parties prepare two random sharings $[a]_t, [b]_t$ in the same way as that in the preparation phase of the DN protocol.
2. All parties compute $[c]_t = [a \cdot b]_t$ using the DN multiplication protocol.

Efficiency Analysis. Note that each step of compression requires $O(k)$ inner-product (or multiplication) operations, which requires $O(kn)$ field elements. Also, each step of compression requires to reconstruct a random sharing, which requires $O(n^2)$ field elements. Therefore, the total amount of communication of verifying m multiplication tuples is $O((kn+n^2)\cdot\log_k m)$ field elements. Since the number of multiplication tuples m is bounded by $\mathrm{poly}(\kappa)$ where κ is the security parameter. If we choose $k = \kappa$, then the cost is just $O(\kappa n + n^2)$ field elements, which is independent of the number of multiplication tuples.

Therefore, the communication complexity per gate of our construction is the same as the DN semi-honest protocol.

Remark 1. An attractive feature of our approach is that the communication cost is not affected by the field size. To see this, note that the cost of our check only has a sub-linear dependence on the circuit size. Therefore, we can run the check over an extension field of the original field with large enough size, which does not influence the concrete efficiency of our construction.

As a comparison, the concrete efficiency of both constructions [CGH$^+$18,NV18] suffer if one uses a small field. This is because in both constructions, the failure probability of the verification depends on the size of the field. For a small field, they need to do the verification several times to acquire the desired security. The same trick does not work because the cost of their checks has a linear dependency on the circuit size.

Remark 2. Compared with the constructions in [CGH$^+$18,NV18], we also remove unnecessary checks to make the protocol as succinct as possible. Specifically, this new technique of verifying a batch of multiplication tuples is the only check in the protocol and the remaining parts are the same as the DN protocol. In particular, we do not check the consistency/validity of the sharings.

3 Preliminaries

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. The communication complexity is measured by the number of bits via private channels between every pair of parties.

We focus on functions that can be represented as arithmetic circuits over a finite field $\mathbb{F}$ (with $|\mathbb{F}| \geq n+1$) with input, addition, multiplication, random, and output gates. Let $\phi = \log |\mathbb{F}|$ be the size of an element in $\mathbb{F}$. We use κ to denote the security parameter and let $\mathbb{K}$ be an extension field of $\mathbb{F}$ (with $|\mathbb{K}| \geq 2^\kappa$). For simplicity, we assume that κ is the size of an element in $\mathbb{K}$.

An adversary is able to corrupt at most $t < n/2$ parties, provide inputs to corrupted parties and receive all messages sent to corrupted parties. Corrupted parties can deviate from the protocol arbitrarily. For simplicity, we assume that $n = 2t + 1$. Let $\mathcal{C}$ denote the set of all corrupted parties and $\mathcal{H}$ denote the set of all honest parties.

Each party P_i is assigned a unique non-zero field element $\alpha_i \in \mathbb{F}\backslash\{0\}$ as the identity. Let c_I, c_M, c_R, c_O be the numbers of input, multiplication, random, and output gates respectively. We set $C = c_I + c_M + c_R + c_O$ to be the size of the circuit.

3.2 Secret Sharing

In our protocol, we use the standard Shamir secret sharing scheme [Sha79].

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

For simplicity, we use $[\boldsymbol{w}]_d$, where $\boldsymbol{w} = (w^{(1)}, w^{(2)}, \ldots, w^{(\ell)}) \in \mathbb{G}^\ell$, to represent a vector of degree-d Shamir sharings $([w^{(1)}]_d, [w^{(2)}]_d, \ldots, [w^{(\ell)}]_d)$.

Properties of the Shamir Secret Sharing Scheme. In the following, we will utilize two properties of the Shamir secret sharing scheme.

- Linear Homomorphism:

$$\forall\ [x]_d, [y]_d,\ [x + y]_d = [x]_d + [y]_d.$$

- Multiplying two degree-d sharings yields a degree-$2d$ sharing. The secret value of the new sharing is the product of the original two secrets.

$$\forall\ [x]_d, [y]_d,\ [x \cdot y]_{2d} = [x]_d \cdot [y]_d.$$

For the first property, we equivalently add the underlying two polynomials. Therefore, the degree remains the same and the secret value becomes the summation of the original two secrets. For the second property, we equivalently multiply the underlying two polynomials. As a result, the degree becomes $2d$ and the secret value is the product of the original two secrets.

Terminologies and Remarks. For a degree-k polynomial $f(\cdot) \in \mathbb{G}[X]$, let $c_0, \ldots, c_k$ denote the coefficients of $f(\cdot)$. If all parties hold degree-d sharings of $c_0, \ldots, c_k$, then for all public input $x \in \mathbb{G}$, all parties can locally compute the degree-d sharing $[f(x)]_d$, which is a linear combination of $[c_0]_d, [c_1]_d, \ldots, [c_k]_d$. Essentially, it means that all parties hold a degree-d sharing of the polynomial $f(\cdot)$. In the following, we use $[f(\cdot)]_d$ to denote a degree-d sharing of the polynomial $f(\cdot)$.

Recall that t is defined to be the upper-bound of the number of corrupted parties. We refer to a pair of sharings $([r]_t, [r]_{2t})$ of the same secret value r as a pair of *double sharings*.

Since $n = 2t + 1$, at least $t + 1$ parties are honest. Therefore, the secret value of a degree-t sharing is determined by the shares held by honest parties. However, when the number of honest parties is larger than $t + 1$, a corrupted party may distribute an invalid degree-t sharing such that the shares held by honest parties are inconsistent. To avoid ambiguity, let $\mathcal{H}$ denote the set of all honest parties and $\mathcal{H}_H \subseteq \mathcal{H}$ be the set of the first $t+1$ honest parties. The secret value of a degree-t sharing is defined to be the secret value of the degree-t sharing determined by the shares held by parties in $\mathcal{H}_H$. We further set $\mathcal{H}_C = \mathcal{H} \backslash \mathcal{H}_H$.

Note that once a degree-t sharing is distributed, the secret value is fixed and in particular, corrupted parties can no longer change the secret value even if the sharing is dealt by a corrupted party.

Remark 3. We point out that the above definition does not address the security issue due to the inconsistent sharings distributed by corrupted parties. However,

it eases the description of the security properties of several semi-honest protocols in the presence of a fully malicious adversary.

Remark 4. Note that the security issue due to the inconsistent sharings only occurs when the number of honest parties is more than $t+1$. Intuitively, this security issue can be tackled by thinking a new adversary that corrupts parties in $\mathcal{C} \bigcup \mathcal{H}_C$ such that parties in $\mathcal{C}$ are controlled by the original adversary and parties in $\mathcal{H}_C$ just faithfully follow the protocol.

3.3 Generating Random Sharings

We introduce a simple protocol RAND, which comes from [DN07], to let all parties prepare $t+1 = O(n)$ random degree-t sharings in the *semi-honest* setting.

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

Protocol 1: RAND

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

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

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

Recall that $\mathcal{H}_H$ is the set of the first $t+1$ honest parties and the secret value of a degree-t sharing is defined to be the secret value of the degree-t sharing determined by the shares held by parties in $\mathcal{H}_H$. We show that the randomness of the secret sharings is preserved in the *fully malicious* setting. We have the following lemma. This lemma is proved in the semi-honest setting in [DN07].

Lemma 1. *Given the views of* RAND *of corrupted parties,* $r^{(1)}, r^{(2)}, \ldots, r^{(t+1)}$ *(the secret values of* $[r^{(1)}]_t, [r^{(2)}]_t, \ldots, [r^{(t+1)}]_t$*) are uniformly random.*

Proof. Let $\boldsymbol{M}^{\mathcal{H}_H}$ denote the sub-matrix of $\boldsymbol{M}$ containing the columns of $\boldsymbol{M}$ with indices in $\mathcal{H}_H$ and $\boldsymbol{M}^{\mathcal{C} \bigcup \mathcal{H}_C}$ denote the sub-matrix of $\boldsymbol{M}$ containing the columns of $\boldsymbol{M}$ with indices in $\mathcal{C} \bigcup \mathcal{H}_C$. Let $(\{s^{(i)}\}_{\mathcal{H}_H})$ denote the vector of the secret values of the sharings dealt by parties in $\mathcal{H}_H$ and $(\{s^{(i)}\}_{\mathcal{C} \bigcup \mathcal{H}_C})$ denote

the vector of the secret values of the sharings dealt by parties in $\mathcal{C} \bigcup \mathcal{H}_C$. Then,

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

Note that $\boldsymbol{M}^{\mathcal{H}_H}$ is a $(t+1) \times (t+1)$ matrix. By the property of Vandermonde matrices, $\boldsymbol{M}^{\mathcal{H}_H}$ is invertible. In RAND, corrupted parties receive at most t shares of the random sharings dealt by parties in $\mathcal{H}_H$. Therefore given the views of Rand of corrupted parties, the secret values of the sharings dealt by parties in $\mathcal{H}_H$ are uniformly random. Since $\boldsymbol{M}^{\mathcal{H}_H}$ is invertible, $\boldsymbol{M}^{\mathcal{H}_H}(\{s^{(i)}\}_{\mathcal{H}_H})^{\mathrm{T}}$ is a vector of uniformly random elements. Therefore, $(r^{(1)}, r^{(2)}, \ldots, r^{(t+1)})$ are uniformly random.

3.4 Generating Random Double Sharings

We introduce a simple protocol DOUBLERAND, which comes from [DN07], to let all parties prepare $t + 1 = O(n)$ pairs of random double sharings in the *semi-honest* setting. The description of DOUBLERAND appears in Protocol 2. The communication complexity of DOUBLERAND is $O(n^2)$ field elements.

Protocol 2: DOUBLERAND

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

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

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

We show that the randomness of the double sharings is preserved in the *fully malicious* setting. We have the following lemma. This lemma is proved in the semi-honest setting in [DN07].

Lemma 2. *Given the views of* DOUBLERAND *of corrupted parties,* $r^{(1)}, \ldots, r^{(t+1)}$ *(the secret values of* $[r^{(1)}]_t, \ldots, [r^{(t+1)}]_t$*) are uniformly random. Also, all shares of* $[r^{(1)}]_{2t}, [r^{(2)}]_{2t}, \ldots, [r^{(t+1)}]_{2t}$ *held by honest parties are uniformly random.*

Proof. Note that compared with the views of RAND of corrupted parties, the only difference is that the views of DOUBLERAND of corrupted parties also contains at most t shares of the random degree-$2t$ sharings dealt by honest parties. By the

security of the Shamir secret sharing scheme, these shares do not leak any information about the secret values. Therefore, we can show that $r^{(1)}, r^{(2)}, \ldots, r^{(t+1)}$ are uniformly random in the same way as that in Lemma 1.

We can think that an honest party P_i randomly samples a pair of double sharings $([s^{(i)}]_t, [s^{(i)}]_{2t})$ in the following way:

1. P_i first randomly samples $n = 2t+1$ elements and sets them to be the shares of a degree-$2t$ sharing.
2. Based on these $2t+1$ random elements, P_i recovers the whole sharing $[s^{(i)}]_{2t}$.
3. P_i randomly samples t elements and sets them to be the shares of a degree-t sharing held by parties in $\mathcal{C} \bigcup \mathcal{H}_C$.
4. P_i recovers the whole sharing $[s^{(i)}]_t$ based on the t shares held by parties in $\mathcal{C} \bigcup \mathcal{H}_C$ and the secret value $s^{(i)}$. The final result is $([s^{(i)}]_t, [s^{(i)}]_{2t})$.

Note that all shares of the degree-$2t$ sharing, and, the shares of the degree-t sharing held by parties in $\mathcal{C} \bigcup \mathcal{H}_C$ are independent and uniformly random elements. Therefore, given the views of DoubleRand of corrupted parties, all shares of $\{[s^{(i)}]_{2t}\}_{i \in \mathcal{H}}$ held by honest parties are independent and uniformly random.

For a degree-$2t$ sharing $[x]_{2t}$, we will use x_i to represent the share held by P_i. Since for an honest party P_j, the shares $r_j^{(1)}, r_j^{(2)}, \ldots, r_j^{(t+1)}$ only depend on the shares of $s_j^{(1)}, s_j^{(2)}, \ldots, s_j^{(n)}$, to show that all shares of $[r^{(1)}]_{2t}, [r^{(2)}]_{2t}, \ldots, [r^{(t+1)}]_{2t}$ held by honest parties are independent and uniformly random, it is sufficient to show that $r_j^{(1)}, r_j^{(2)}, \ldots, r_j^{(t+1)}$ are uniformly random. Note that $\{s_j^{(i)}\}_{i \in \mathcal{H}}$ are uniformly random and

$$(r_j^{(1)}, r_j^{(2)}, \ldots, r_j^{(t+1)})^{\mathrm{T}} = \boldsymbol{M}(s_j^{(1)}, s_j^{(2)}, \ldots, s_j^{(n)})^{\mathrm{T}}.$$

Therefore, we can show that $r_j^{(1)}, r_j^{(2)}, \ldots, r_j^{(t+1)}$ are uniformly random in the same way as that in Lemma 1.

4 Extensions of the DN Multiplication Protocol and the Batch-wise Multiplication Verification Technique

4.1 Extension of the DN Multiplication Protocol

In this part, we introduce a natural extension to the DN Multiplication Protocol [DN07]. We first introduce the basic protocol, which takes two input sharings $[x]_t, [y]_t$ and outputs $[x \cdot y]_t$. The protocol needs to consume a pair of double sharings. In the whole protocol, all parties will first call DoubleRand to prepare the double sharings.

The description of the DN Multiplication Protocol (denoted by Mult) appears in Protocol 3. The communication complexity of Mult is $O(n)$ field elements.

We point out that Mult is a *semi-honest* protocol. However, we will show that the privacy of the input sharings $[x]_t, [y]_t$ is preserved in the *fully malicious* setting. We have the following lemma.

Protocol 3: MULT

1. All parties agree on a special party $P_{\texttt{king}}$. Let $([r]_t, [r]_{2t})$ be the random double sharings which will be used in the protocol.
2. All parties locally compute $[x \cdot y + r]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$.
3. $P_{\texttt{king}}$ collects all shares and reconstructs the secret value $x \cdot y + r$. Then $P_{\texttt{king}}$ randomly generates a degree-t sharing $[x \cdot y + r]_t$ and distributes the shares to other parties.
4. All parties locally compute $[x \cdot y]_t = [x \cdot y + r]_t - [r]_t$.

Lemma 3. *Given the views of* DOUBLERAND *of corrupted parties, the shares sent from honest parties to* $P_{\texttt{king}}$ *are uniformly random and independent of* $[x]_t, [y]_t$.

Proof. According to Lemma 2, the shares of $[r]_{2t}$ held by honest parties are uniformly random elements given the views of DOUBLERAND of corrupted parties. Therefore, the shares of $[x \cdot y + r]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$ held by honest parties are uniformly random.

In essence, the DN Multiplication Protocol does a degree reduction from $[x \cdot y]_{2t} = [x]_t \cdot [y]_t$ to $[x \cdot y]_t$. Let $\odot$ denote the inner-product operation. For two vectors of degree-t sharings $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t$ of dimension ℓ, to compute $[\boldsymbol{x} \odot \boldsymbol{y}]_t$, we can first compute $[\boldsymbol{x} \odot \boldsymbol{y}]_{2t} = [\boldsymbol{x}]_t \odot [\boldsymbol{y}]_t$ and then use the same idea as the DN Multiplication Protocol to compute $[\boldsymbol{x} \odot \boldsymbol{y}]_t$ from $[\boldsymbol{x} \odot \boldsymbol{y}]_{2t}$. In this way, the cost is just one multiplication operation. This idea has been observed in several previous works and in particular, has been used in [CGH+18] to design an MPC protocol for a small field.

The description of the extended DN Multiplication Protocol (denoted by EXTEND-MULT) appears in Protocol 4. The communication complexity of EXTEND-MULT is $O(n)$ field elements.

Protocol 4: EXTEND-MULT

1. All parties agree on a special party $P_{\texttt{king}}$. Let $([r]_t, [r]_{2t})$ be the random double sharings which will be used in the protocol.
2. All parties locally compute $[\boldsymbol{x} \odot \boldsymbol{y} + r]_{2t} = [\boldsymbol{x}]_t \odot [\boldsymbol{y}]_t + [r]_{2t}$.
3. $P_{\texttt{king}}$ collects all shares and reconstructs the secret value $\boldsymbol{x} \odot \boldsymbol{y} + r$. Then $P_{\texttt{king}}$ randomly generates a degree-t sharing $[\boldsymbol{x} \odot \boldsymbol{y} + r]_t$ and distributes the shares to other parties.
4. All parties locally compute $[\boldsymbol{x} \odot \boldsymbol{y}]_t = [\boldsymbol{x} \odot \boldsymbol{y} + r]_t - [r]_t$.

Similarly, we will show that the privacy of the input sharings $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t$ is preserved in the *fully malicious* setting. We have the following lemma.

Lemma 4. *Given the views of* DoubleRand *of corrupted parties, the shares sent from honest parties to* $P_{\texttt{king}}$ *are uniformly random and independent of* $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t$.

Remark 5. We note that Extend-Mult can be further extended so that given $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, \boldsymbol{c})$, all parties can compute a degree-t sharing of $\sum_{i=1}^{\ell} c_i \cdot x^{(i)} \cdot y^{(i)}$. To see this, all parties can first compute $[\boldsymbol{x}']_t = (c_1[x^{(1)}]_t, c_2[x^{(2)}]_t, \ldots, c_\ell[x^{(\ell)}]_t)$. Then invoke Extend-Mult on $[\boldsymbol{x}']_t$ and $[\boldsymbol{y}]_t$ to get the desired degree-t sharing.

4.2 Extension of the Batch-wise Multiplication Verification Technique

In this part, we introduce a natural extension to the Batch-wise Multiplication Verification Technique [BSFO12]. We first introduce the basic technique, which is used to check the correctness of a batch of multiplication tuples efficiently.

Overview of the Batch-wise Multiplication Verification Technique. For simplicity, suppose that we are working on a large enough finite field $\mathbb{G}$. Given m multiplication tuples

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

we want to check whether $x^{(i)} \cdot y^{(i)} = z^{(i)}$ for all $i \in [m]$.

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

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

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

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

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

Therefore, to check whether $f \cdot g = h$, all parties first prepare a random degree-t sharing $[r]_t$ by invoking Rand. Then all parties open the sharing by

sending their shares to all other parties. All parties locally compute $[f(r)]_t, [g(r)]_t$ and $[h(r)]_t$. In the case that $|\mathbb{G}|$ is large enough (say $|\mathbb{G}| = 2^\kappa$ where κ is the security parameter), it is sufficient to only check whether $([f(r)], [g(r)]_t, [h(r)]_t)$ is a correct multiplication tuple since we accept errors with negligible probability.

Checking the Single Multiplication Tuple. In [BSFO12], this check is done using an "expensive" MPC protocol. Since the number of checks is independent of the number of original multiplication tuples we need to check, the cost of this check does not affect the overall communication complexity. In [NV18], a random multiplication tuple is included when using the Batch-wise Multiplication Verification technique (so that the technique applies on $m + 1$ multiplication tuples). In this way, revealing the whole sharings $([f(r)]_t, [g(r)]_t, [h(r)]_t)$ does not compromise the security of the original multiplication tuples. Therefore, all parties simply send their shares of $[f(r)]_t, [g(r)]_t, [h(r)]_t$ to all other parties and then check whether $f(r) \cdot g(r) = h(r)$.

Description of Compress. In essence, this technique compresses m checks of multiplication tuples into 1 check of a single tuple. The protocol takes m multiplication tuples as input and outputs a single tuple. We refer to this protocol as Compress. The description of Compress appears in Protocol 5. The communication complexity of Compress is $O(mn + n^2)$ field elements.

Protocol 5: Compress

1. Let $[r]_t$ be the random sharing which will be used in the protocol. The multiplication tuples are denoted by

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

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

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

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

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

   All parties locally compute $[h(\cdot)]_t$ by using $\{[z^{(i)}]_t\}_{i\in[2m-1]}$.
5. All parties send their shares of $[r]_t$ to other parties to reconstruct r. If $r \in [m]$, all parties abort. Otherwise, output $([f(r)]_t, [g(r)]_t, [h(r)]_t)$.

Lemma 5. *The probability that all parties abort in* Compress *is* $m/|\mathbb{G}|$. *Also, if at least one multiplication tuple is incorrect, then the resulting tuple output by* Compress *is incorrect with probability* $1 - 2(m-1)/|\mathbb{G}|$.

Proof. According to Lemma 1, r is uniformly random. Note that the abortion occurs only when $r \in [m]$. Therefore the probability that all parties abort is $m/|\mathbb{G}|$.

If there exists an incorrect multiplication tuple, then $f \cdot g \neq h$. Since the polynomial $h - f \cdot g$ is a degree-$2(m-1)$ non-zero polynomial, the number of $x \in \mathbb{G}$ such that $h(x) - f(x) \cdot g(x) = 0$ is at most $2(m-1)$. Therefore, with probability $1 - 2(m-1)/|\mathbb{G}|$, $(f(r), g(r), h(r))$ is incorrect.

Extension. A natural extension of the Batch-wise Multiplication Verification technique is to check the correctness of m inner-product tuples. This idea has been observed in [NV18]. However, this extension is not used in the main result of [NV18].

Given m inner-product tuples

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

where $\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \in \mathbb{G}^\ell$ for all $i \in [m]$, we want to check whether $\boldsymbol{x}^{(i)} \odot \boldsymbol{y}^{(i)} = z^{(i)}$ for all $i \in [m]$. The idea is to construct two vectors of degree-$(m-1)$ polynomials $\boldsymbol{f}(\cdot), \boldsymbol{g}(\cdot)$ such that

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

All parties can locally compute $[\boldsymbol{f}(\cdot)]_t$ and $[\boldsymbol{g}(\cdot)]_t$ by using $\{[\boldsymbol{x}^{(i)}]_t\}_{i\in[m]}$ and $\{[\boldsymbol{y}^{(i)}]_t\}_{i\in[m]}$ respectively.

For $i \in \{m+1, \ldots, 2m-1\}$, all parties compute $[\boldsymbol{f}(i)]_t, [\boldsymbol{g}(i)]_t$, and then compute the degree-t sharing $[z^{(i)}]_t$ by invoking Extend-Mult on $([\boldsymbol{f}(i)]_t, [\boldsymbol{g}(i)]_t)$. Let $h(\cdot)$ be a degree-$2(m-1)$ polynomial such that

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

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

The remaining steps are similar to that in Compress. We refer to this extension as Extend-Compression. The description of Extend-Compress appears in Protocol 6. The communication complexity of Extend-Compress is $O(mn + n^2)$ field elements.

Lemma 6. *The probability that all parties abort in* Extend-Compress *is* $m/|\mathbb{G}|$. *Also, if at least one inner-product tuple is incorrect, then the resulting tuple output by* Extend-Compress *is incorrect with probability* $1 - 2(m-1)/|\mathbb{G}|$.

Remark 6. We note that the field $\mathbb{G}$ should contain at least $2m-1$ elements. Otherwise the polynomial $h(\cdot)$ is not well-defined. However, the condition that $|\mathbb{G}| = 2^\kappa$ can be relaxed without blowing up the failure probability. The main observation is that a polynomial $f(\cdot) \in \mathbb{G}$ is also a valid polynomial in an extension field of $\mathbb{G}$. We can choose a large enough extension field $\tilde{\mathbb{G}}$ of $\mathbb{G}$ and generate the random sharing $[r]_t$ in $\tilde{\mathbb{G}}$. In this way, the failure probability only depends on the size of the extension field and is independent of the size of $\mathbb{G}$.

Protocol 6: Extend-Compress

1. Let $[r]_t$ be the random sharing which will be used in the protocol. The inner-product tuples are denoted by
$$([\boldsymbol{x}^{(1)}]_t, [\boldsymbol{y}^{(1)}]_t, [z^{(1)}]_t), ([\boldsymbol{x}^{(2)}]_t, [\boldsymbol{y}^{(2)}]_t, [z^{(2)}]_t), \ldots, ([\boldsymbol{x}^{(m)}]_t.[\boldsymbol{y}^{(m)}]_t, [z^{(m)}]_t).$$
2. Let $\boldsymbol{f}(\cdot), \boldsymbol{g}(\cdot)$ be vectors of degree-$(m-1)$ polynomials such that
$$\forall i \in [m], \boldsymbol{f}(i) = \boldsymbol{x}^{(i)}, \boldsymbol{g}(i) = \boldsymbol{y}^{(i)}.$$
All parties locally compute $[\boldsymbol{f}(\cdot)]_t$ and $[\boldsymbol{g}(\cdot)]_t$ by using $\{[\boldsymbol{x}^{(i)}]_t\}_{i\in[m]}$ and $\{[\boldsymbol{y}^{(i)}]_t\}_{i\in[m]}$ respectively.
3. For all $i \in \{m+1, \ldots, 2m-1\}$, all parties locally compute $[\boldsymbol{f}(i)]_t$ and $[\boldsymbol{g}(i)]_t$, and then invoke Extend-Mult on $([\boldsymbol{f}(i)]_t, [\boldsymbol{g}(i)]_t)$ to compute $[z^{(i)}]_t = [\boldsymbol{f}(i) \odot \boldsymbol{g}(i)]_t$.
4. Let $h(\cdot)$ be a degree-$2(m-1)$ polynomials such that
$$\forall i \in [2m-1], h(i) = z^{(i)}.$$
All parties locally compute $[h(\cdot)]_t$ by using $\{[z^{(i)}]_t\}_{i\in[2m-1]}$.
5. All parties send their shares of $[r]_t$ to other parties to reconstruct r. If $r \in [m]$, all parties abort. Otherwise, output $([\boldsymbol{f}(r)]_t, [\boldsymbol{g}(r)]_t, [h(r)]_t)$.

5 Multiplication Verification

In this section, we introduce our new method to efficiently verify a batch of multiplication tuples. We refer the readers to Section 2 for a high-level idea of our method.

5.1 Step One: De-Linearization

The first step is to transform the check of m multiplication tuples into one check of an inner-product tuple of dimension m. The description of De-Linearization appears in Protocol 7. The communication complexity of De-Linearization is $O(n^2)$ elements in $\mathbb{K}$.

Lemma 7. *If at least one multiplication tuple is incorrect, then the resulting inner-product tuple output by* De-Linearization *is also incorrect with overwhelming probability.*

5.2 Step Two: Dimension-Reduction

The second step is to reduce the dimension of the inner-product tuple output by De-Linearization. We will use Extend-Compress as a building block. The

Protocol 7: De-Linearization

1. Let $[r]_t$ be the random sharing which will be used in the protocol. Here $r \in \mathbb{K}$. The multiplication tuples are denoted by

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

2. All parties send their shares of $[r]_t$ to all other parties.
3. All parties reconstruct the value r. Then set

$$\begin{aligned}
[\boldsymbol{x}]_t &= ([x^{(1)}]_t, r[x^{(2)}]_t, \ldots, r^{m-1}[x^{(m)}]_t) \\
[\boldsymbol{y}]_t &= ([y^{(1)}]_t, [y^{(2)}]_t, \ldots, [y^{(m)}]_t) \\
[z]_t &= \sum_{i=1}^{m} r^{i-1}[z^{(i)}]_t,
\end{aligned}$$

and output $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$.

description of Dimension-Reduction appears in Protocol 8. The communication complexity of Dimension-Reduction is $O(kn + n^2)$ elements in $\mathbb{K}$, where k is the compression parameter.

Lemma 8. *If the input inner-product tuple is incorrect, then the resulting inner-product tuple output by* Dimension-Reduction *is also incorrect with overwhelming probability.*

5.3 Step Three: Randomization

In the final step, we add a random multiplication tuple when we use Compress so that the verification of the resulting multiplication tuple can be done by simply opening all the sharings. The description of Randomization appears in Protocol 9. The communication complexity of Randomization is $O(mn + n^2)$ elements in $\mathbb{K}$, where m is the dimension of the inner-product tuple.

Lemma 9. *If the input inner-product tuple is incorrect, then at least one honest party will take* `fail` *as output with overwhelming probability.*

6 Protocol

In this section, we show how to use our new technique to construct a secure-with-abort protocol. We will first give the protocols to handle input gates in Section 6.1, addition gates and multiplication gates in Section 6.2 and output gates in Section 6.3. Then we show how to check the correctness of all multiplication gates in Section 6.4. Finally, we give the main protocol in Section 6.5.

Protocol 8: DIMENSION-REDUCTION

1. The inner-product tuple is denoted by $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$. Let k denote the compression parameter and m denote the dimension of the inner-product tuple (i.e., the dimension of the vector $\boldsymbol{x}$). Let $\ell = m/k$.
2. All parties interpret $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t$ as

$$[\boldsymbol{x}]_t = ([\boldsymbol{a}^{(1)}]_t, [\boldsymbol{a}^{(2)}]_t, \ldots, [\boldsymbol{a}^{(k)}]_t)$$
$$[\boldsymbol{y}]_t = ([\boldsymbol{b}^{(1)}]_t, [\boldsymbol{b}^{(2)}]_t, \ldots, [\boldsymbol{b}^{(k)}]_t),$$

   where $\{\boldsymbol{a}^{(i)}, \boldsymbol{b}^{(i)}\}_{i\in[k]}$ are vectors of dimension ℓ.
3. For $i \in [k-1]$, all parties invoke EXTEND-MULT on $([\boldsymbol{a}^{(i)}]_t, [\boldsymbol{b}^{(i)}]_t)$ to compute $[c^{(i)}]_t$ where $c^{(i)} = \boldsymbol{a}^{(i)} \odot \boldsymbol{b}^{(i)}$. Then set

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

4. All parties invoke EXTEND-COMPRESS on

$$([\boldsymbol{a}^{(1)}]_t, [\boldsymbol{b}^{(1)}]_t, [c^{(1)}]_t), ([\boldsymbol{a}^{(2)}]_t, [\boldsymbol{b}^{(2)}]_t, [c^{(2)}]_t), \ldots, ([\boldsymbol{a}^{(k)}]_t, [\boldsymbol{b}^{(k)}]_t, [c^{(k)}]_t).$$

   The output is denoted by $([\boldsymbol{a}]_t, [\boldsymbol{b}]_t, [c]_t)$. All parties take this new inner-product tuple as output.

6.1 Input Gates

In this part, we handle the input gates. For each input x, the input holder P_i simply generates a random degree-t sharing of x and distributes the shares to other parties. The description of INPUT appears in Protocol 10. The communication complexity of INPUT is $O(c_I n)$ field elements in $\mathbb{F}$, where c_I is the number of inputs.

Lemma 10. *The messages sent from honest parties to corrupted parties only contain uniform elements.*

6.2 Addition Gates and Multiplication Gates

In this part, we handle the addition gates and multiplication gates. For each addition gate with input sharings $[x]_t, [y]_t$, all parties locally compute the resulting sharing $[z]_t = [x]_t + [y]_t$. For each multiplication gate with input sharings $[x]_t, [y]_t$, all parties invoke MULT to compute the resulting sharing $[z]_t$ where $z = x \cdot y$. The description of EVAL appears in Protocol 11. The communication complexity of EVAL is $O(c_M n)$ field elements in $\mathbb{F}$, where c_M is the number of multiplication gates.

Protocol 9: RANDOMIZATION

1. Let $[a^{(0)}]_t, [b^{(0)}]_t$ be the two random degree-t sharings which will be used in the protocol. The inner-product tuple is denoted by $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$. Let m denote the dimension of the inner-product tuple.
2. All parties interpret $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t$ as
$$[\boldsymbol{x}]_t = ([a^{(1)}]_t, [a^{(2)}]_t, \ldots, [a^{(m)}]_t)$$
$$[\boldsymbol{y}]_t = ([b^{(1)}]_t, [b^{(2)}]_t, \ldots, [b^{(m)}]_t).$$
3. All parties invoke MULT on $([a^{(0)}]_t, [b^{(0)}]_t)$ to compute $[c^{(0)}]_t$ where $c^{(0)} = a^{(0)} \cdot b^{(0)}$.
4. For $i \in [m-1]$, all parties invoke MULT on $([a^{(i)}]_t, [b^{(i)}]_t)$ to compute $[c^{(i)}]_t$ where $c^{(i)} = a^{(i)} \cdot b^{(i)}$. Then set
$$[c^{(m)}]_t = [z]_t - \sum_{i=1}^{m-1} [c^{(i)}]_t.$$
5. All parties invoke COMPRESS on
$$([a^{(0)}]_t, [b^{(0)}]_t, [c^{(0)}]_t), ([a^{(1)}]_t, [b^{(1)}]_t, [c^{(1)}]_t), \ldots, ([a^{(m)}]_t, [b^{(m)}]_t, [c^{(m)}]_t).$$
The output is denoted by $([a]_t, [b]_t, [c]_t)$.
6. All parties send their shares of $[a]_t, [b]_t, [c]_t$ to all other parties.
7. All parties reconstruct a, b, c. For each party P_i, if either the shares of $[a]_t, [b]_t, [c]_t$ are inconsistent or $a \cdot b \neq c$, P_i takes `fail` as output. Otherwise, P_i takes `ok` as output.

Protocol 10: INPUT

1. For each input x, the input holder P_i randomly samples a degree-t sharing $[x]_t$ and distributes the shares to all other parties.

Protocol 11: EVAL

1. For each input gate with input sharings $[x]_t, [y]_t$, all parties locally compute $[z]_t = [x]_t + [y]_t$.
2. For each multiplication gate with input sharings $[x]_t, [y]_t$, all parties invoke MULT on $[x]_t, [y]_t$ to compute $[z]_t$ where $z = x \cdot y$.

Lemma 11. *The messages sent from honest parties to corrupted parties only contain uniform elements.*

Proof. Note that addition gates do not require any communication. Therefore all messages are sent in the invocations of Mult. It follows from Lemma 3.

6.3 Output Gates

In this part, we handle the output gates. For each output gate, let $[x]_t$ denote the sharing associated with this gate and P_i be the party who should receive this output. P_i simply collects shares from all other parties and reconstructs the result. The description of Output appears in Protocol 12. The communication complexity of Output is $O(c_O n)$ field elements in $\mathbb{F}$. where c_O is the number of outputs.

Protocol 12: Output

1. For each output gate, let $[x]_t$ be the sharing associated with this gate and P_i be the party who should receive this output.
2. P_i collects all shares of $[x]_t$ and reconstructs the result x.

6.4 Multiplication Verification

In this section, we show how to check the correctness of all multiplication gates. It is a simple combination of De-Linearization, Dimension-Reduction, Randomization introduced in Section 5. The description of MultVerification appears in Protocol 13.

Now we analyze the communication complexity of MultVerification. Recall that each time of running Dimension-Reduction reduces the dimension of the inner-product tuple to be $1/k$ of the original dimension. Therefore, MultVerification includes 1 invocation of De-Linearization, $(\log_k c_M - 1)$ invocations of Dimension-Reduction and 1 invocation of Randomization. The communication complexity of MultVerification is

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

field elements in $\mathbb{K}$.

Remark 7. We note that the circuit size is bounded by $\text{poly}(\kappa)$ where κ is the security parameter. Therefore, if we set $k = \kappa$, the communication complexity of MultVerification becomes $O(n\kappa + n^2)$ field elements in $\mathbb{K}$.

Remark 8. Note that MultVerification requires $O(\log_k c_M)$ rounds. In the real world, one can adjust k based on the overhead of each round and the overhead of sending each bit via a private channel to achieve the best running time.

Protocol 13: MULTVERIFICATION

1. Let k be the compression parameter. Recall that c_M is the number of multiplication gates in the circuit. The multiplication tuples are denoted by
$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(c_M)}]_t.[y^{(c_M)}]_t, [z^{(c_M)}]_t).$$
2. All parties invoke DE-LINEARIZATION on these c_M multiplication tuples. Let $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$ denote the output.
3. While the dimension of $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$ is larger than k, all parties invoke DIMENSION-REDUCTION and set
$$([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t) := \text{DIMENSION-REDUCTION}(([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t), k).$$
4. All parties invoke RANDOMIZATION on $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$.

6.5 Main Protocol

In this part, we give the main protocol which achieves security-with-abort. The description of MAIN appears in Protocol 14. The communication complexity of MAIN is $O(Cn\phi + \log_k C \cdot (kn + n^2)\kappa)$ *bits*, where C is the circuit size, k is the compression parameter, ϕ is the size of an field element in $\mathbb{F}$ and κ is the security parameter.

Theorem 1. *Protocol 14 is secure-with-abort against fully malicious adversaries in the presence of honest majority.*

The proof is given in Appendix A.

Analysis of the Concrete Efficiency. We point out that, without MULTVERIFICATION, MAIN is the same as the best-known semi-honest protocol [DN07]. The cost per multiplication gate is 6 field elements in $\mathbb{F}$ per party, including 4 field elements to prepare a pair of random double sharings, 1 element sending to $P_{\texttt{king}}$, 1 element receiving from $P_{\texttt{king}}$. Note that the cost of MULTVERIFICATION is bounded by $O(\log_k C(kn + n^2)\kappa)$ bits, which does not influence the cost per multiplication gate. Therefore, MAIN achieves the same concrete efficiency as the best-known semi-honest protocol [DN07].

6.6 An Optimization of DN Multiplication Protocol

We note that since $P_{\texttt{king}}$ can potentially be corrupted in MULT, there is no need to protect the secrecy of the sharing distributed by $P_{\texttt{king}}$. Recall that $P_{\texttt{king}}$ needs to generate and distribute a degree-t sharing $[x \cdot y + r]_t$. Since any t shares of a degree-t sharing are independent of its secret value, $P_{\texttt{king}}$ can predetermine t shares to be 0 and still generate a valid degree-t sharing of $[x \cdot y + r]_t$. In

Protocol 14: MAIN

1. In the following, if a party receives an inconsistent sharing, then this party aborts.
2. **Preparation Phase**:
   (i) All parties invoke DOUBLERAND $c_M/(t+1)$ times to generate enough random double sharings in $\mathbb{F}$ which will be used for multiplication gates.
   (ii) All parties invoke RAND $c_R/(t+1)$ times, where c_R is the number of random gates in C, to generate enough random sharings in $\mathbb{K}$ for random gates.
   (iii) All parties invoke RAND $(\log_k C+3)/(t+1)$ times to generate enough random sharings in $\mathbb{K}$ which will be used in EXTEND-COMPRESS, COMPRESS, DE-LINEARIZATION and RANDOMIZATION when verifying the correctness of all multiplication tuples.
   (iv) All parties invoke DOUBLERAND $2k\log_k C/(t+1)$ times to generate enough random double sharings in $\mathbb{K}$ which will be used in MULT and EXTEND-MULT when verifying the correctness of all multiplication tuples.
3. **Input Phase**: All parties invoke INPUT.
4. **Computation Phase**:
   (i) All parties invoke EVAL.
   (ii) All parties invoke MULTVERIFICATION. If a party takes `fail` as output, this party aborts.
5. **Output Phase**: If no party aborts, all parties invoke OUTPUT.

this way, however, a party whose share is 0 automatically learns it without any communication.

In more detail, all parties first choose a set of $t+1$ parties (including $P_{\texttt{king}}$). Let $\mathcal{T}$ denote the set of these $t+1$ parties. $P_{\texttt{king}}$ first sets the shares held by parties outside of $\mathcal{T}$ to be 0. Then use these t shares and the secret value to recover the shares held by parties in $\mathcal{T}$. $P_{\texttt{king}}$ only distributes the shares to parties in $\mathcal{T}$. The description of OPT-MULT appears in Protocol 15.

The concrete efficiency of OPT-MULT is 5.5 field elements per party. This trick can also be used in EXTEND-MULT. After applying this optimization, the concrete efficiency of our protocol reduces to 5.5 field elements per gate (per party).

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.

Protocol 15: OPT-MULT

1. All parties agree on a special party $P_{\texttt{king}}$. Let $\mathcal{T}$ be a set of $t+1$ parties (including $P_{\texttt{king}}$) all parties agree on. Let $([r]_t, [r]_{2t})$ be the random double sharings which will be used in the protocol.
2. All parties locally compute $[x \cdot y + r]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$.
3. $P_{\texttt{king}}$ collects all shares and reconstructs the secret value $x \cdot y + r$. Then $P_{\texttt{king}}$ sets the shares of parties outside of $\mathcal{T}$ to be 0. $P_{\texttt{king}}$ recovers the whole sharing $[x \cdot y + r]_t$ using these t shares of 0 and the secret value $x \cdot y + r$.
4. $P_{\texttt{king}}$ distributes the shares of $[x \cdot y + r]_t$ to parties in $\mathcal{T}$. The parties outside of $\mathcal{T}$ set their shares to be 0.
5. All parties locally compute $[x \cdot y]_t = [x \cdot y + r]_t - [r]_t$.

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

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.

BTH06. Zuzana Beerliova-Trubiniova and Martin Hirt. Efficient multi-party computation with dispute control. In *Theory of Cryptography Conference*, pages 305–328. Springer, 2006.

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.

CDD+99. Ronald Cramer, Ivan Damgård, Stefan Dziembowski, Martin Hirt, and Tal Rabin. Efficient multiparty computations secure against an adaptive adversary. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 311–326, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

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.

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.

FL19. Jun Furukawa and Yehuda Lindell. Two-thirds honest-majority mpc for malicious adversaries at almost the cost of semi-honest. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS ?19, page 1557?1571, New York, NY, USA, 2019. Association for Computing Machinery.

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.

GIP+14a. Daniel Genkin, Yuval Ishai, Manoj M. Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 495–504, New York, NY, USA, 2014. ACM.

GIP+14b. Daniel Genkin, Yuval Ishai, Manoj M Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 495–504. ACM, 2014.

GLS19. Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional mpc with guaranteed output delivery. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 85–114, Cham, 2019. Springer International Publishing.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

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.

IKP+16. Yuval Ishai, Eyal Kushilevitz, Manoj Prabhakaran, Amit Sahai, and Ching-Hua Yu. Secure protocol transformations. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 430–458, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

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.

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.

NV18. Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority mpc by batchwise multiplication verification. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security*, pages 321–339, Cham, 2018. Springer International Publishing.

RBO89. Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85. ACM, 1989.

Sha79. Adi Shamir. How to share a secret. *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 Proof of the Security

In this section, we formally prove Theorem 1. We first show how to construct a simulator $\mathcal{S}$ in Appendix A.1 and then show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties with overwhelming probability in Appendix A.2.

A.1 Construction of the Simulator

In this part, we construct a simulator $\mathcal{S}$ which will be used to prove Theorem 1 in the next part.

Recall that in Section 3, $\mathcal{H}$ is the set of all honest parties and $\mathcal{C}$ is the set of all corrupted parties. We further set $\mathcal{H}_H \subseteq \mathcal{H}$ to be the set of the first $t+1$ honest parties and $\mathcal{H}_C = \mathcal{H} \backslash \mathcal{H}_H$. Recall that the secret value of a degree-t sharing is defined to be the secret value of the degree-t sharing determined by the shares held by parties in $\mathcal{H}_H$.

General Strategy. We note that except for the input phase, the local computation of an honest party only depends on the shares this party received from other parties, and in particular, is independent of its input. For parties in $\mathcal{H}_C$, the simulator $\mathcal{S}$ will honestly follow the protocol except for the input phase. Specifically, it can be achieved by explicitly generate the messages sent from parties in $\mathcal{H}_H$ to parties in $\mathcal{H}_C$. In this way, parties in $\mathcal{H}_C$ will receive all messages it requires to follow the protocol. Therefore we only need to focus on simulating the behaviors of the parties in $\mathcal{H}_H$.

We will show that the protocol maintains the invariance that all degree-t sharings all parties hold are known linear combinations of degree-t sharings generated by each party. Concretely, each degree-t sharing $[x]_t$ can be represented by

$$[x]_t = \sum_{i=1}^{n} [x(i)]_t,$$

where $[x(i)]_t$ is either directly dealt by P_i or a known linear combination of several degree-t sharings dealt by P_i. It means that P_i knows the whole sharing $[x(i)]_t$.

Now we show how $\mathcal{S}$ can simulate the behaviors of honest parties by utilizing these two facts.

Computing the Shares held by Parties in $\mathcal{H}_H$. In the protocol, all parties need to reconstruct several degree-t sharings by sending their shares to other parties. $\mathcal{S}$ needs to explicitly generate the shares held by parties in $\mathcal{H}_H$. Suppose $[x]_t$ is the sharing all parties need to reconstruct and $\mathcal{S}$ knows the value x.

According to the invariance, we have $[x]_t = \sum_{i=1}^{n} [x(i)]_t$. We further set

$$[x(\mathcal{C})]_t = \sum_{P_i \in \mathcal{C}} [x(i)]_t,$$
$$[x(\mathcal{H}_C)]_t = \sum_{P_i \in \mathcal{H}_C} [x(i)]_t,$$
$$[x(\mathcal{H}_H)]_t = \sum_{P_i \in \mathcal{H}_H} [x(i)]_t.$$

Note that $\mathcal{S}$ receives from corrupted parties the shares of $[x(\mathcal{C})]_t$ held by parties in $\mathcal{H}_H$, and therefore can reconstruct the value $x(\mathcal{C})$. As for $[x(\mathcal{H}_C)]_t$, the whole sharing has been honestly generated and distributed by $\mathcal{S}$. Therefore $\mathcal{S}$ also knows $x(\mathcal{H}_C)$. The only task is to compute the shares of $[x(\mathcal{H}_H)]_t$ held by honest parties.

$\mathcal{S}$ computes $x(\mathcal{H}_H) = x - x(\mathcal{C}) - x(\mathcal{H}_C)$. For $[x(\mathcal{H}_H)]_t$, the shares held by parties in $\mathcal{C} \bigcup \mathcal{H}_C$ have been explicitly generated and distributed by $\mathcal{S}$. Note that $|\mathcal{C} \bigcup \mathcal{H}_C| = t$. Based on these t shares and the secret value $x(\mathcal{H}_H)$, $\mathcal{S}$ can compute the shares of $[x(\mathcal{H}_H)]_t$ held by parties in $\mathcal{H}_H$.

We point out that this strategy is not affected by inconsistent sharings dealt by corrupted parties. I.e., even if the shares of $[x(\mathcal{C})]_t$ held by honest parties $(\mathcal{H}_C \bigcup \mathcal{H}_H)$ are inconsistent, $\mathcal{S}$ still perfectly simulates the behaviors of honest parties. This is because what $\mathcal{S}$ does is computing the shares of $[x(\mathcal{H}_H)]_t$ held by parties in $\mathcal{H}_H$, which are determined by the secret value $x(\mathcal{H}_H)$ and the shares $\mathcal{S}$ sent to parties in $\mathcal{C} \bigcup \mathcal{H}_C$, and therefore, is independent of the sharings dealt by corrupted parties.

Simulating Mult. According to Lemma 3, $\mathcal{S}$ only needs to send uniformly random and independent field elements to $P_{\texttt{king}}$ when Mult is invoked. However, $\mathcal{S}$ also needs to know how much the difference is caused by the behaviors of corrupted parties. It will be used when simulating Compress and Extend-Compress.

Specifically, let $[x]_t, [y]_t$ be the input sharings of the multiplication gate and $([r]_t, [r]_{2t})$ be the double sharings used in Mult. For parties in $\mathcal{H}_C$, $\mathcal{S}$ honestly computes and sends the shares to $P_{\texttt{king}}$. For parties in $\mathcal{H}_H$, $\mathcal{S}$ chooses uniformly random elements and sends them to $P_{\texttt{king}}$. Note that $\mathcal{S}$ will receive from $P_{\texttt{king}}$ the shares of $[x \cdot y + r]_t$ held by parties in $\mathcal{H}_H$. Therefore $\mathcal{S}$ can reconstruct the value $x \cdot y + r$ dealt by $P_{\texttt{king}}$. Now we need to compute the value $x \cdot y + r$ it should be, i.e., when corrupted parties behave honestly.

Recall that in DoubleRand, $([r]_t, [r]_{2t})$ is a linear combination of the double sharings generated by each party. Therefore, we have

$$([r]_t, [r]_{2t}) = \sum_{i=1}^{n} ([r(i)]_t, [r(i)]_{2t}),$$

where $([r(i)]_t, [r(i)]_{2t})$ is dealt by P_i. For $P_i \in \mathcal{C} \bigcup \mathcal{H}_C$, $\mathcal{S}$ can compute $r(i)$ from the shares of $[r(i)]_t$ held by parties in $\mathcal{H}_H$. Therefore, the only task is to learn the value $x \cdot y + \sum_{P_i \in \mathcal{H}_H} r(i)$.

To this end, $\mathcal{S}$ will compute all shares of $[x]_t \cdot [y]_t + \sum_{P_i \in \mathcal{H}_H} [r(i)]_{2t}$ and then reconstruct the secret value. For parties in $\mathcal{H}_H$, note that $\mathcal{S}$ has chosen the shares of $[x \cdot y + r]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$ held by parties in $\mathcal{H}_H$. Since for all $P_i \in \mathcal{C} \bigcup \mathcal{H}_C$, $\mathcal{S}$ received from P_i the shares of $[r(i)]_{2t}$ held by parties in $\mathcal{H}_H$, $\mathcal{S}$ can compute the shares of $[x]_t \cdot [y]_t + \sum_{P_i \in \mathcal{H}_H} [r(i)]_{2t}$ held by parties in $\mathcal{H}_H$.

As for parties in $\mathcal{C} \bigcup \mathcal{H}_C$, $\mathcal{S}$ first computes the shares of $[x]_t$ they should hold. Recall that we can decompose $[x]_t$ into three parts $[x(\mathcal{C})]_t, [x(\mathcal{H}_C)]_t, [x(\mathcal{H}_H)]_t$. For $[x(\mathcal{C})]_t, [x(\mathcal{H}_C)]_t$, $\mathcal{S}$ learns the shares held by parties in $\mathcal{H}_H$. Based on these shares, $\mathcal{S}$ can compute the shares that parties in $\mathcal{C} \bigcup \mathcal{H}_C$ should hold. For $[x(\mathcal{H}_H)]_t$, the shares held by parties in $\mathcal{C} \bigcup \mathcal{H}_C$ are chosen and distributed by $\mathcal{S}$. Therefore, $\mathcal{S}$ can compute the shares of $[x]_t$ that parties in $\mathcal{C} \bigcup \mathcal{H}_C$ should hold. Similarly, $\mathcal{S}$ computes the shares of $[y]_t$ they should hold. Note that for $P_i \in \mathcal{H}_H$, the shares of $[r(i)]_{2t}$ held by parties in $\mathcal{C} \bigcup \mathcal{H}_C$ are chosen and distributed by $\mathcal{S}$. Therefore, $\mathcal{S}$ can compute the shares of $[x]_t \cdot [y]_t + \sum_{P_i \in \mathcal{H}_H} [r(i)]_{2t}$ that parties in $\mathcal{C} \bigcup \mathcal{H}_C$ should hold.

After computing all shares of $[x]_t \cdot [y]_t + \sum_{P_i \in \mathcal{H}_H} [r(i)]_{2t}$, $\mathcal{S}$ computes the secret value $x \cdot y + \sum_{P_i \in \mathcal{H}_H} r(i)$ and then computes the value $x \cdot y + r$. Finally, $\mathcal{S}$ computes the difference caused by the behaviors of corrupted parties.

Detailed Simulation. Now we describe the behavior of the simulator $\mathcal{S}$.

Simulating the Preparation Phase. In the preparation phase, we need to simulate the behaviors of honest parties in DoubleRand and Rand.

For parties in $\mathcal{H}_C$, $\mathcal{S}$ faithfully generates and distributes random double sharings for DoubleRand or random degree-t sharings for Rand. For parties in $\mathcal{H}_H$, $\mathcal{S}$ sends to parties in $\mathcal{C} \bigcup \mathcal{H}_C$ uniform elements as the shares of random double sharings for DoubleRand or random degree-t sharings for Rand. Note that $|\mathcal{C} \bigcup \mathcal{H}_C| = t$ and any t shares of a random degree-t sharing or a random degree-$2t$ sharing are uniformly random and independent of the secret value. Therefore, $\mathcal{S}$ perfectly simulates the behaviors of honest parties.

Regarding the invariance, note that in DoubleRand, each pair of double sharings is a linear combination of double sharings dealt by each party. In particular, each degree-t sharing is a linear combination of degree-t sharings dealt by each party. Therefore, the invariance is maintained. Similarly, we can show that the invariance holds for the random degree-t sharings generated in Rand.

Simulating the Input Phase. In the input phase, we need to simulate the behaviors of honest parties in Input.

$\mathcal{S}$ sends uniform elements to parties in $\mathcal{C} \bigcup \mathcal{H}_C$ on behalf of each honest party. Note that any t shares of a degree-t sharing are uniformly random and independent of the secret value. Therefore $\mathcal{S}$ perfectly simulates the behaviors of honest parties.

$\mathcal{S}$ computes the inputs of corrupted parties based on the shares held by parties in $\mathcal{H}_H$.

Note that each degree-t sharing generated in Input is dealt by a single party, i.e., the input holder. The invariance is maintained.

Simulating the Computation Phase – Eval. In this part, we need to simulate the behaviors of honest parties in Eval.

In Eval, no communication is required for addition gates. As for a multiplication gate, $\mathcal{S}$ can simulate the behaviors of honest parties in the way mentioned in the last part and compute the difference caused by the behaviors of corrupted parties.

Regarding the invariance, the invariance holds for the resulting degree-t sharing of an addition gate since it holds for the two input sharings. For a multiplication gate, the resulting degree-t sharing is $[x \cdot y + r]_t - [r]_t$ where $[x \cdot y + r]_t$ is dealt by $P_{\texttt{king}}$ and $[r]_t$ is a random degree-t sharing generated in Rand. Therefore the invariance is maintained.

Simulating the Computation Phase – MultVerification. As for MultVerification, we describe the strategy of $\mathcal{S}$ for De-Linearization, Dimension-Reduction and Randomization separately.

- For De-Linearization, the only communication is reconstructing the random degree-t sharing $[r]_t$. According to Lemma 1, r should be a uniform element. $\mathcal{S}$ randomly samples $r \in \mathbb{K}$ then computes the shares held by parties in $\mathcal{H}_H$ in the way mentioned in the last part. $\mathcal{S}$ honestly computes the shares held by parties in $\mathcal{H}_C$. Then $\mathcal{S}$ distributes the shares held by honest parties to other parties.
  After r is reconstructed, $\mathcal{S}$ computes the difference of the resulting inner-product tuple $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$, i.e., $z - \boldsymbol{x} \odot \boldsymbol{y}$, which can be computed by using the differences from the original multiplication tuples. Note that these differences have been computed when $\mathcal{S}$ simulated the multiplication gates.
- For Dimension-Reduction, $\mathcal{S}$ simulates the behaviors of honest parties in Extend-Mult in a similar way to that in Mult. $\mathcal{S}$ also computes the difference for each inner-product tuple caused by the behaviors of corrupted parties. When invoking Extend-Compress, Extend-Mult can be simulated in the same way as we just mentioned, and reconstructing $[r]_t$ can be simulated in the same way as that in De-Linearization.
  After r is reconstructed, $\mathcal{S}$ computes the difference of the resulting tuple using the differences of the input inner-product tuples.
- For Randomization, Mult and Compress can be simulated in similar ways to those in Extend-Mult and Extend-Compress. Since a and b are linear combinations of $a^{(0)}, a^{(1)}, \ldots, a^{(m)}$ and $b^{(0)}, b^{(1)}, \ldots, b^{(m)}$ respectively and $a^{(0)}, b^{(0)}$ are uniformly random, a and b are also uniformly random.
  $\mathcal{S}$ randomly samples $a, b \in \mathbb{K}$ and computes c based on a, b and the difference for the resulting tuple. Then $\mathcal{S}$ computes the shares of $[a]_t, [b]_t, [c]_t$ held by parties in $\mathcal{H}_H$ in the same way mentioned in the last part. $\mathcal{S}$ honestly computes the shares held by parties in $\mathcal{H}_C$. Finally, $\mathcal{S}$ distributes the shares held by honest parties to other parties.

Regarding the invariance, note that all parties just apply additions and multiplications on the sharings they hold. With the same argument as that for Eval, the invariance is maintained.

Simulating the Output Phase. In this part, we need to simulate the behaviors of honest parties in OUTPUT.

$\mathcal{S}$ first invokes the ideal functionality with the inputs of corrupted parties computed when simulating INPUT.

In OUTPUT, for each output gate with $[x]_t$ associating with it, $\mathcal{S}$ computes the shares of $[x]_t$ held by parties in $\mathcal{H}_H$ in the way mentioned in the last part. $\mathcal{S}$ honestly computes the shares held by parties in $\mathcal{H}_C$. Finally, $\mathcal{S}$ sends the shares held by honest parties to the party who should receive this result.

This finishes the description of the simulator $\mathcal{S}$.

A.2 Proof

In this part, we prove Theorem 1.

Theorem 1. *Protocol 14 is secure-with-abort against fully malicious adversaries in the presence of honest majority.*

Proof. We show that the view of an adversary $\mathcal{A}$ when interacting with the simulator $\mathcal{S}$ we constructed in Section A.1 has the same distribution as that in the real world with all but a negligible probability. Consider the following hybrids.

$\mathbf{Hybrid_0}$: Execution in the real world.

$\mathbf{Hybrid_1}$: During the input phase, $\mathcal{S}$ computes the inputs of corrupted parties based on the shares held by parties in $\mathcal{H}_H$. Then, in the output phase, $\mathcal{S}$ invokes the ideal functionality and compares the result and the secret values of the output sharings. If they do not match, $\mathcal{S}$ aborts.

Note that $\mathcal{S}$ simply checks the correctness of the protocol. According to Lemma 7, Lemma 8 and Lemma 9, the computation is correct with overwhelming probability. Therefore, the distribution is statistical close to $\mathbf{Hybrid_0}$.

$\mathbf{Hybrid_2}$: In this hybrid, $\mathcal{S}$ simulates the output phase. Note that the behaviors of parties in $\mathcal{H}_C$ remain the same. The shares of the output sharings held by parties in $\mathcal{H}_H$ are determined by the result of the functionality and the shares sent or received by parties in $\mathcal{H}_H$ before the output phase. Therefore, the distribution is the same as $\mathbf{Hybrid_1}$.

$\mathbf{Hybrid_3}$: In this hybrid, $\mathcal{S}$ computes the difference of all multiplication tuples and inner-product tuples. When invoking RANDOMIZATION, the secret value of $[c]_t$ is set to be $a \cdot b$ plus the corresponding difference. The shares of $[c]_t$ held by honest parties are prepared by $\mathcal{S}$.

Note that the secret value of $[c]_t$ is the same as the original value. Since the behaviors of parties in $\mathcal{H}_C$ remain the same and the shares of $[c]_t$ held by parties in $\mathcal{H}_H$ are determined by c and the shares sent or received by parties in $\mathcal{H}_H$ before, the distribution is the same as $\mathbf{Hybrid_2}$.

$\mathbf{Hybrid_4}$: In this hybrid, MULTVERIFICATION is replaced by the simulation of $\mathcal{S}$. Note that the operations that require interaction are EXTEND-MULT, MULT and opening random degree-t sharings. According to Lemma 4, Lemma 3 and Lemma 1, $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Therefore, the distribution is the same as $\mathbf{Hybrid_3}$.

Hybrid$_5$: In this hybrid, EVAL is replaced by the simulation of $\mathcal{S}$. Note that the operation that requires interaction is MULT. According to Lemma 3, $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Therefore, the distribution is the same as **Hybrid$_4$**.

Hybrid$_6$: In this hybrid, DOUBLERAND and RAND in the preparation phase are replaced by the simulation of $\mathcal{S}$. According to Lemma 2 and Lemma 1, $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Therefore, the distribution is the same as **Hybrid$_5$**.

Hybrid$_7$: In this hybrid, INPUT is replaced by the simulation of $\mathcal{S}$. According to Lemma 10, $\mathcal{S}$ perfectly simulates the behaviors of honest parties. Therefore, the distribution is the same as **Hybrid$_6$**.

Note that **Hybrid$_7$** is the execution between $\mathcal{S}$ and $\mathcal{A}$ in the ideal world. We conclude that the distribution of **Hybrid$_7$** is statistical close to **Hybrid$_0$**.