# 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.

## 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$^+$14,LN17,CGH$^+$18,NV18] have focused on improving the communication complexity. Genkin et al. [GIP$^+$14] provided the first construction with malicious security (with abort) having the same asymptotic communication complexity as the best-known semi-honest protocol [DN07] (hereafter referred to as the DN protocol). Since then, the main focus in this line of research has been to improve the concrete communication complexity per gate. Compared with the DN protocol, the recent breakthrough [CGH$^+$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

V. Goyal—Research supported in part by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via 2019-1902070008, an NSF award 1916939, a gift from Ripple, a JP Morgan Faculty Fellowship, a PNC center for financial services innovation award, and a Cylab seed funding award.
Y. Song—Research supported in part by a Cylab Presidential Fellowship and grants of Vipul Goyal mentioned above.

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 of 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 obtained by building on the technique in [BBCG$^+$19]. We observe that the additional cost in [CGH$^+$18,NV18] comes from the verification of the multiplications. We introduce a new technique inspired by [BBCG$^+$19], which allows us to bring 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$^+$18,NV18], we also remove several checks to make the protocol as succinct as possible. Specifically, the verification a batch of multiplication tuples is the only check in the protocol and the remaining parts are the same as the semi-honest DN protocol. In particular, we do not check the consistency/validity of the sharings as required in [CGH$^+$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, $\phi$ denote the size of a field element, $\kappa$ denote the security parameter, and $n$ denote the number of parties participating in the computation.

*Security with abort.* In [DN07], Damgård and Nielsen introduce the best-known semi-honest protocol, which we refer to as the DN protocol. The communication complexity of the DN protocol is $O(Cn\phi)$ bits. The concrete efficiency is 6 field elements per multiplication gate (per party). In [GIP$^+$14], Genkin, et al. show that the DN protocol is secure up to an additive attack when running in the fully malicious setting. Based on this observation, a secure-with-abort MPC protocol can be constructed by combining the DN protocol and a circuit which is resilient to an additive attack (referred to as an AMD circuit). As a result, Genkin, et al. [GIP$^+$14] give the first construction against a fully malicious adversary with communication complexity $O(Cn\phi)$ bits (for a large enough field), which matches the asymptotic communication complexity of the DN protocol.

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

An interesting observation is that the theorem showed in [GIP$^+$14] implies that the DN protocol provides perfect privacy of honest parties (before the output phase) in the presence of a fully malicious adversary. To achieve security with abort, the only task is to check the correctness of the computation before the output phase. This observation has been used in [LN17,NV18]. In particular, the construction in [NV18] achieves the same concrete efficiency as [CGH$^+$18] by using the Batch-wise Multiplication Verification technique in [BSFO12], i.e., 12 field elements per multiplication gate. Our construction also relies on this observation. Therefore, the main task is to efficiently verify a batch of multiplications such that the communication complexity is sublinear in the number of parties.

In [BBCG$^+$19], Boneh, et al. introduce a very powerful tool to achieve this task when the number of parties is restricted to be a constant. Our result is obtained by instantiating this technique with a different secret sharing scheme, which allows us to overcome this restriction so that it works for any (polynomial) number of parties. Furthermore, we simplify this technique by avoiding the use of a robust secret sharing scheme and a verifiable secret sharing scheme, which are required in [BBCG$^+$19]. Our protocol additionally makes a simple optimization to the DN protocol, which brings down the cost from 6 field elements per multiplication gate to 5.5 field elements. More details about the comparison for techniques can be found in the last paragraph of Section 2.6. A subsequent work [GLOS20] implements our construction and shows that the performance beats the previously best-known implementation result [CGH$^+$18] in this setting.

In [BGIN19], Boyle, et al. use the technique in [BBCG$^+$19] to construct a 3-party computation with guaranteed output delivery. In particular, they implement their verification for multiplication gates. As shown in their implementation result, just the local computation of checking the correctness of 1 million multiplication gates in the 31-bit Mersenne Field requires around 1 second. Note that this does not include any computation cost related to the circuit and any communication cost. On the other hand, the implementation result from [GLOS20] shows that our construction only needs 0.7 second for computing the whole circuit in an even large field (61-bit Mersenne Field) in the 3-party setting. This shows that our construction is *several* times faster.

*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$^+$14] showed that one can even achieve sub-constant cost per gate relying on packed secret sharing. Several works have also focused on the performance of 2-party computation and 3-party computation in practice. Examples include [LP12,NNOB12] for 2-party computation, [FLNW17,ABF$^+$17] for 3-party computation and so on.

## 2 Technical Overview

### 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$^+$14], 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

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

where $\{\boldsymbol{a}^{(i)}, \boldsymbol{b}^{(i)}\}_{i \in [k]}$ are vectors of dimension $\ell$. 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 $\ell$.

*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 poly($\kappa$) where $\kappa$ 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.

*Relation with the Technique in [BBCG+19].* We note that our idea is similar to the technique in [BBCG+19] when it is used to construct MPC protocols. When $n = 3$ and $t = 1$, our construction is very similar to the construction in [BBCG+19]. For a general $n$-party setting, the construction in [BBCG+19] relies on the replicated secret sharings and builds upon the sublinear distributed zero knowledge proofs constructed in [BBCG+19]. However, the computation cost of the replicated secret sharings goes exponentially in the number of parties. This restricts the construction in [BBCG+19] to only work for a constant number of parties. On the other hand, we explore the use of the Shamir secret sharing scheme in the $n$-party setting.

Our idea is inspired by the extensions of the DN multiplication protocol [DN07,CGH$^+$18] and the Batch-wise Multiplication Verification [BSFO12,NV18]. This allows us to get a positive result without relying on replicated secret sharings. We also note that the construction in [BBCG$^+$19] requires the sharings (related to the distributed zero knowledge proof) to be robust and verifiable. We simplify this technique by removing the use of a robust secret sharing scheme and a verifiable secret sharing scheme.

Moreover, we explore a recursion trick to further improve the communication complexity of verifying multiplications. Compared with the construction in [BBCG$^+$19] which requires to communicate $O(\sqrt{C})$ bits, we achieve $O((kn + n^2) \cdot \log_k C \cdot \kappa)$ bits. Our protocol additionally makes a simple optimization to the DN protocol, which brings down the cost from 6 field elements per multiplication to 5.5 field elements.

## 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 $\kappa$ 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 $\kappa$ 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.

*Client-Server Model.* For the simplicity of the proof, we will show the security of our protocol in the *client-server* model. In the client-server model, clients provide inputs to the functionality and receive outputs, and servers can participate in the computation but do not have inputs or get outputs. Each party may have different roles in the computation. Note that, if every party plays a single client and a single server, this corresponds to a protocol in the standard MPC model. Let $c$ denote the number of clients and $n = 2t + 1$ denote the number of servers. We will show that our construction is secure against a fully malicious adversary controlling at most $c$ clients and $t$ servers.

One benefit of the client-server model is the following theorem shown in [GIP$^+$14].

**Theorem 1 (Lemma 5.2 [GIP$^+$14]).** *Let $\Pi$ be a protocol computing an c-client circuit $C$ using $n = 2t+1$ servers. Then, if $\Pi$ is secure against any adversary controlling exactly $t$ servers, then $\Pi$ is secure against any adversary controlling at most $t$ servers.*

This theorem allows us to only consider the case where the adversary controls exactly $t$ servers. Therefore, in the following, we assume that there are exactly $t$ corrupted servers.

In our construction, the clients only participate in the input phase and the output phase. The main computation is conducted by the servers. For simplicity, we continue to use $\{P_1, \ldots, P_n\}$ for the $n$ servers, and refer to the servers as parties.

### 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 $f(\cdot)$.

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

## 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 functionality is presented in Functionality 1.

**Functionality 1:** $\mathcal{F}_{\text{rand}}$

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

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

We show that this protocol securely computes Functionality 1 in the presence of *a fully malicious* adversary.

**Protocol 2:** 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 $[r^{(1)}]_t, [r^{(2)}]_t, \ldots, [r^{(t+1)}]_t$.

**Lemma 1.** *The protocol* Rand *securely computes the functionality $\mathcal{F}_{rand}$ in the presence of a fully malicious adversary controlling $t$ corrupted parties.*

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

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

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

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

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

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

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

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

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

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

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

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

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

Note that $\boldsymbol{M}^{\mathcal{H}}$ is a $(t+1) \times (t+1)$ matrix. By the property of Vandermonde matrices, $\boldsymbol{M}^{\mathcal{H}}$ is invertible. Therefore, given the sharings $\{[s^{(i)}]_t\}_{i \in \mathcal{C}}$ dealt by corrupted parties, there is a one-to-one map from $\{[s^{(i)}]_t\}_{i \in \mathcal{H}}$ to $\{[r^{(i)}]_t\}_{i \in [t+1]}$. Note that the only difference between $\mathbf{Hybrid}_1$ and $\mathbf{Hybrid}_2$ is that, in

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

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

## 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. Recall that a pair of double sharings $([r]_t, [r]_{2t})$ contains two sharings of the same secret value $r$. Double sharings will be used to evaluate multiplication gates.

The functionality is presented in Functionality 3. The description of the protocol DoubleRand appears in Protocol 4. The communication complexity of DoubleRand is $O(n^2)$ field elements.

**Functionality 3:** $\mathcal{F}_{\text{doubleRand}}$

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

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

**Protocol 4:** 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

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

   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 this protocol securely computes Functionality 3 in the presence of *a fully malicious* adversary.

**Lemma 2.** *The protocol* DoubleRand *securely computes the functionality* $\mathcal{F}_{doubleRand}$ *in the presence of a fully malicious adversary controlling* $t$ *corrupted parties.*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

generated by $\mathcal{F}_{\text{doubleRand}}$, we can compute back to a set of valid double sharings $\{([s^{(i)}]_t, [s^{(i)}]_{2t})\}_{i\in\mathcal{H}}$. Therefore, the distribution of $\mathbf{Hybrid}_2$ is identical to the distribution of $\mathbf{Hybrid}_1$.

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

## 3.5 Generating Random Coins

Relying on $\mathcal{F}_{\text{rand}}$, we show how to securely generate a random field element. The functionality $\mathcal{F}_{\text{coin}}$ is presented in Functionality 5. The description of Coin appears in Protocol 6. The communication complexity of Coin is $O(n^2)$ field elements.

**Functionality 5:** $\mathcal{F}_{\text{coin}}$

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

**Protocol 6:** Coin

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

**Lemma 3.** *The protocol* Coin *securely computes $\mathcal{F}_{coin}$ with abort in the $\mathcal{F}_{rand}$-hybrid model in the presence of a fully malicious adversary controlling $t$ corrupted parties.*

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

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

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

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

## 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 $[x \cdot y]_t$. In [GIP$^+$14], Genkin et al. prove that the semi-honest DN protocol is secure up to an additive attack in the presence of a fully malicious adversary. An additive attack means that the adversary is able to change the multiplication result $x \cdot y$ by adding an arbitrary fixed value $d$ to $x \cdot y + d$. Therefore, in the functionality $\mathcal{F}_{\text{mult}}$, we allow the adversary to change the result by sending $d$ to $\mathcal{F}_{\text{mult}}$. The functionality $\mathcal{F}_{\text{mult}}$ also sends the shares of $[x]_t, [y]_t$ held by corrupted parties to the adversary. Note that these shares are known to the adversary. The functionality $\mathcal{F}_{\text{mult}}$ is presented in Functionality 7.

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

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

The protocol needs to consume a pair of random double sharings, which is prepared by calling $\mathcal{F}_{\text{doubleRand}}$. The description of the DN Multiplication Protocol (denoted by Mult) appears in Protocol 8. The communication complexity of Mult is $O(n)$ field elements.

**Protocol 8:** Mult

1. All parties agree on a special party $P_{\texttt{king}}$. Let $[x]_t, [y]_t$ denote the input sharings.
2. All parties invoke $\mathcal{F}_{\text{doubleRand}}$ to prepare a pair of random double sharings $([r]_t, [r]_{2t})$.
3. All parties locally compute $[x \cdot y + r]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$.
4. $P_{\texttt{king}}$ 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.
5. All parties locally compute $[x \cdot y]_t = [x \cdot y + r]_t - [r]_t$.

**Lemma 4.** *The protocol* Mult *securely computes the functionality $\mathcal{F}_{mult}$ in the $\mathcal{F}_{doubleRand}$-hybrid model in the presence of a fully malicious adversary controlling $t$ corrupted parties.*

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

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

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

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

*Simulation of* Mult. In the beginning, $\mathcal{S}$ receives from $\mathcal{F}_{\text{mult}}$ the shares of $[x]_t, [y]_t$ held by corrupted parties. When invoking $\mathcal{F}_{\text{doubleRand}}$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{doubleRand}}$ and receives from the adversary the shares of $[r]_t, [r]_{2t}$ held by corrupted parties.

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

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

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

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

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

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

Note that the shares of $[x \cdot y]_t$ held by honest parties are determined by the shares of corrupted parties and the secret, which is determined by the multiplication result $x \cdot y$ and the difference $d$. Therefore, the distribution of **Hybrid**$_1$ is the same as **Hybrid**$_0$.

**Hybrid**$_2$: In this hybrid, $\mathcal{S}$ simulates honest parties in the whole protocol Mult. Note that the only difference is that, in **Hybrid**$_1$, $\mathcal{S}$ uses the real shares of $[x \cdot y + r]_{2t}$ of honest parties, while in **Hybrid**$_2$, $\mathcal{S}$ generates random elements as the shares of $[x \cdot y + r]_{2t}$ of honest parties. However, as we showed in the beginning, the shares of $[r]_{2t}$ held by honest parties are uniformly random. Therefore, the shares of $[x\cdot y+r]_{2r}$ held by honest parties are also uniformly random. Thus, the distribution of **Hybrid**$_2$ is the same as **Hybrid**$_1$.

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

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 $\ell$, 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 functionality $\mathcal{F}_{\text{extendMult}}$ appears in Functionality 9, and the description of the extended DN Multiplication Protocol (denoted by EXTEND-MULT) appears in Protocol 10. The communication complexity of EXTEND-MULT is $O(n)$ field elements.

**Functionality 9:** $\mathcal{F}_{\text{extendMult}}$

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

**Protocol 10:** EXTEND-MULT

1. All parties agree on a special party $P_{\texttt{king}}$. Let $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t$ denote the input vectors of sharings.
2. All parties invoke $\mathcal{F}_{\text{doubleRand}}$ to prepare a pair of random double sharings $([r]_t, [r]_{2t})$.
3. All parties locally compute $[\boldsymbol{x} \odot \boldsymbol{y} + r]_{2t} = [\boldsymbol{x}]_t \odot [\boldsymbol{y}]_t + [r]_{2t}$.
4. $P_{\texttt{king}}$ 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.
5. All parties locally compute $[\boldsymbol{x} \odot \boldsymbol{y}]_t = [\boldsymbol{x} \odot \boldsymbol{y} + r]_t - [r]_t$.

**Lemma 5.** *The protocol* EXTEND-MULT *securely computes the functionality $\mathcal{F}_{extendMult}$ in the $\mathcal{F}_{doubleRand}$-hybrid model in the presence of a fully malicious adversary controlling $t$ corrupted parties.*

This lemma can be proved in the same way as that for Lemma 4. Therefore, for simplicity, we omit the details.

*Remark 3.* 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 $\mathcal{F}_{\text{mult}}$ 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 invoke $\mathcal{F}_{\text{coin}}$ to generate a random challenge $r$. All parties locally compute $[f(r)]_t, [g(r)]_t$ and $[h(r)]_t$. In the case that $|\mathbb{G}|$ is large enough (say $|\mathbb{G}| = 2^\kappa$ where $\kappa$ 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 11. The communication complexity of Compress is $O(mn + n^2)$ field elements.

**Lemma 6.** *If at least one multiplication tuple is incorrect, then the resulting tuple output by* Compress *is incorrect with probability* $1 - \frac{3m-2}{|\mathbb{G}|}$.

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

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

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

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

**Protocol 11:** COMPRESS

1. The multiplication tuples are denoted by
$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(m)}]_t.[y^{(m)}]_t, [z^{(m)}]_t).$$
2. Let $f(\cdot), g(\cdot)$ be degree-$(m-1)$ polynomials such that
$$\forall i \in [m], f(i) = x^{(i)}, g(i) = y^{(i)}.$$
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 $\mathcal{F}_{\text{mult}}$ 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 invoke $\mathcal{F}_{\text{coin}}$ to generate a random field element $r$. If $r \in [m]$, all parties abort. Otherwise, output $([f(r)]_t, [g(r)]_t, [h(r)]_t)$.

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 $\mathcal{F}_{\text{extendMult}}$ 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 12. The communication complexity of EXTEND-COMPRESS is $O(mn + n^2)$ field elements.

**Lemma 7.** *If at least one inner-product tuple is incorrect, then the resulting tuple output by* EXTEND-COMPRESS *is incorrect with probability* $1 - \frac{3m-2}{|\mathbb{G}|}$.

This lemma can be proved in the same way as that for Lemma 6. Therefore, for simplicity, we omit the details.

*Remark 4.* 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 12:** Extend-Compress

1. 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 $\mathcal{F}_{\text{extendMult}}$ 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 invoke $\mathcal{F}_{\text{coin}}$ to generate a random field element $r$. 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 13. The communication complexity of De-Linearization is $O(n^2)$ elements in $\mathbb{K}$.

**Protocol 13:** De-Linearization

1. The multiplication tuples are denoted by
$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(m)}]_t.[y^{(m)}]_t, [z^{(m)}]_t).$$
2. All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random field element $r \in \mathbb{K}$.
3. All parties 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 $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$.

**Lemma 8.** *If at least one multiplication tuple is incorrect, then the resulting inner-product tuple output by* De-Linearization *is also incorrect with overwhelming probability.*

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

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

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

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

Since $r$ is randomly sampled by $\mathcal{F}_{\text{coin}}$, the probability that $\mathcal{F}_{\text{coin}}$ outputs such a bad $r$ is bounded by $\frac{3m-2}{|\mathbb{K}|}$. Recall that $\kappa$ is the security parameter and the size of $\mathbb{K}$ is at least $2^\kappa$. Therefore, with probability $1 - \frac{3m-2}{|\mathbb{K}|} \geq 1 - \frac{3m-2}{2^\kappa}$, the tuple output by Compress is incorrect. □

## 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 description of Dimension-Reduction appears in Protocol 14. The communication complexity of Dimension-Reduction is $O(kn + n^2)$ elements in $\mathbb{K}$, where $k$ is the compression parameter.

**Protocol 14:** 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 $\ell$.
3. For $i \in [k-1]$, all parties invoke $\mathcal{F}_{\text{extendMult}}$ 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.

**Lemma 9.** *If the input inner-product tuple is incorrect, then the resulting inner-product tuple output by* Dimension-Reduction *is also incorrect with overwhelming probability.*

*Proof.* Suppose that the input inner-product tuple $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$ is incorrect. We first show that at least one of the following $k$ inner-product tuples is incorrect:

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

Note that the first $k-1$ tuples are computed via $\mathcal{F}_{\text{extendMult}}$. If at least one of the inner-product tuples in the first $k-1$ tuples is incorrect, then the statement holds. Assume that the first $k-1$ tuples are all correct, i.e., for all $i \in [k-1]$, $\boldsymbol{a}^{(i)} \odot \boldsymbol{b}^{(i)} = c^{(i)}$. Since the input inner-product tuple is incorrect, we have $\boldsymbol{x} \odot \boldsymbol{y} \neq z$. Therefore

$$\boldsymbol{a}^{(k)} \odot \boldsymbol{b}^{(k)} = \boldsymbol{x} \odot \boldsymbol{y} - \sum_{i=1}^{k-1} \boldsymbol{a}^{(i)} \odot \boldsymbol{b}^{(i)} = \boldsymbol{x} \odot \boldsymbol{y} - \sum_{i=1}^{k-1} c^{(i)} \neq z - \sum_{i=1}^{k-1} c^{(i)} = c^{(k)},$$

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

According to Lemma 7, the resulting tuple $([\boldsymbol{a}]_t, [\boldsymbol{b}]_t, [c]_t)$ output by Extend-Compress is incorrect with probability $1 - \frac{3k-2}{|\mathbb{K}|} \geq 1 - \frac{3k-2}{2^\kappa}$. $\square$

## 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 15. The communication complexity of Randomization is $O(mn+n^2)$ elements in $\mathbb{K}$, where $m$ is the dimension of the inner-product tuple.

**Protocol 15:** Randomization

1. 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 two times of $\mathcal{F}_{\text{rand}}$ to prepare two random degree-$t$ sharings $[a^{(0)}]_t, [b^{(0)}]_t$.
4. All parties invoke $\mathcal{F}_{\text{mult}}$ on $([a^{(0)}]_t, [b^{(0)}]_t)$ to compute $[c^{(0)}]_t$ where $c^{(0)} = a^{(0)} \cdot b^{(0)}$.
5. For $i \in [m-1]$, all parties invoke $\mathcal{F}_{\text{mult}}$ on $([a^{(i)}]_t, [b^{(i)}]_t)$ to compute $[c^{(i)}]_t$ where $c^{(i)} = a^{(i)} \cdot b^{(i)}$. Then set

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

6. 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 $([a]_t, [b]_t, [c]_t)$.
7. All parties send their shares of $[a]_t, [b]_t, [c]_t$ to all other parties.
8. All parties reconstruct $a, b, c$. For each party $P_i$, if the shares of $[a]_t, [b]_t, [c]_t$ are inconsistent or $a \cdot b \neq c$, $P_i$ aborts. Otherwise, $P_i$ takes `accept` as output.

**Lemma 10.** *If the input inner-product tuple is incorrect, then at least one honest party will either abort or take* `reject` *as output with overwhelming probability.*

*Proof.* Suppose that the input inner-product tuple $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$ is incorrect. We first show that at least one of the following $m$ multiplication tuples is incorrect:

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

Note that the first $m-1$ tuples are computed via $\mathcal{F}_{\text{mult}}$. If at least one of the multiplication tuples in the first $m-1$ tuples is incorrect, then the statement holds. Assume that the first $m-1$ tuples are all correct, i.e., for all $i \in [m-1]$, $a^{(i)} \odot b^{(i)} = c^{(i)}$. Since the input inner-product tuple is incorrect, we have $\boldsymbol{x} \odot \boldsymbol{y} \neq z$. Therefore

$$a^{(m)} \cdot b^{(m)} = \boldsymbol{x} \odot \boldsymbol{y} - \sum_{i=1}^{m-1} a^{(i)} \cdot b^{(i)} = \boldsymbol{x} \odot \boldsymbol{y} - \sum_{i=1}^{m-1} c^{(i)} \neq z - \sum_{i=1}^{m-1} c^{(i)} = c^{(m)},$$

which means that the last multiplication tuple must be incorrect.

According to Lemma 6, the resulting tuple $([a]_t, [b]_t, [c]_t)$ output by Compress is incorrect with probability $1 - \frac{3m+1}{|\mathbb{K}|} \geq 1 - \frac{3m+1}{2^\kappa}$. Note that an incorrect tuple will cause at least one honest party aborting or taking `reject` as output. □

## 5.4 Summary: Multiplication Verification with Sub-linear Communication

In this section, we show how to check the correctness of $m$ multiplication tuples with communication complexity $o(m)$. It is a simple combination of the three protocols De-Linearization, Dimension-Reduction, and Randomization. The functionality $\mathcal{F}_{\text{multVerify}}$ is introduced in Functionality 16. The description of MultVerification appears in Protocol 17.

**Functionality 16:** $\mathcal{F}_{\text{multVerify}}$

1. Let $m$ denote the number of multiplication tuples. The multiplication tuples are denoted by
$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(m)}]_t . [y^{(m)}]_t, [z^{(m)}]_t).$$
2. For all $i \in [m]$, $\mathcal{F}_{\text{multVerify}}$ receives from honest parties their shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$. Then $\mathcal{F}_{\text{multVerify}}$ reconstructs the secrets $x^{(i)}, y^{(i)}, z^{(i)}$. $\mathcal{F}_{\text{multVerify}}$ further computes the shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$ held by corrupted parties and sends these shares to the adversary.
3. For all $i \in [m]$, $\mathcal{F}_{\text{multVerify}}$ computes $d^{(i)} = z^{(i)} - x^{(i)} \cdot y^{(i)}$ and sends $d^{(i)}$ to the adversary.
4. Finally, let $b \in \{\texttt{abort}, \texttt{accept}\}$ denote whether there exists $i \in [m]$ such that $d^{(i)} \neq 0$. $\mathcal{F}_{\text{multVerify}}$ sends $b$ to the adversary and waits for its response.
   - If the adversary replies `continue`, $\mathcal{F}_{\text{multVerify}}$ sends $b$ to honest parties.
   - If the adversary replies `abort`, $\mathcal{F}_{\text{multVerify}}$ sends `abort` to honest parties.

**Lemma 11.** *The protocol* MultVerification *securely computes $\mathcal{F}_{multVerify}$ with abort in the $\{\mathcal{F}_{coin}, \mathcal{F}_{mult}, \mathcal{F}_{extendMult}\}$-hybrid model in the presence of a fully malicious adversary controlling $t$ corrupted parties.*

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

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

**Protocol 17:** MultVerification

1. Let $k$ be the compression parameter, $m$ denote the number of multiplication tuples. The multiplication tuples are denoted by
$$([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), ([x^{(2)}]_t, [y^{(2)}]_t, [z^{(2)}]_t), \ldots, ([x^{(m)}]_t.[y^{(m)}]_t, [z^{(m)}]_t).$$
2. All parties invoke De-Linearization on these $m$ multiplication tuples. Let $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$ 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 $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$.

– Simulation of De-Linearization:
When $\mathcal{F}_{\text{coin}}$ is invoked in Step 2, $\mathcal{S}$ emulates $\mathcal{F}_{\text{coin}}$ and generates a random field element $r \in \mathbb{K}$. Then, $\mathcal{S}$ computes the shares of $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t$ held by corrupted parties and the difference $d = z - \boldsymbol{x} \odot \boldsymbol{y}$. This can be achieved by using the shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$ held by corrupted parties and the difference $d^{(i)} = z^{(i)} - x^{(i)} \cdot y^{(i)}$ for all $i \in [m]$.

– Simulation of Dimension-Reduction:
We will maintain the invariance that, for the input inner-product tuple $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$, $\mathcal{S}$ learns the shares held by corrupted parties and the difference $d = z - \boldsymbol{x} \odot \boldsymbol{y}$. Note that this is true for the first time of invocation of Dimension-Reduction since these shares and the difference are computed when simulating De-Linearization.
In Step 2 and Step 3, $\mathcal{S}$ computes the shares of $[\boldsymbol{a}^{(i)}]_t, [\boldsymbol{b}^{(i)}]_t, [c^{(i)}]_t$ held by corrupted parties and the difference $d^{(i)} = c^{(i)} - \boldsymbol{a}^{(i)} \odot \boldsymbol{b}^{(i)}$ for all $i \in [k]$. Concretely,
- For all $i \in [k]$, the shares of $[\boldsymbol{a}^{(i)}]_t, [\boldsymbol{b}^{(i)}]_t$ held by corrupted parties can be directly obtained from the shares of $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t$ held by corrupted parties.
- For all $i \in [k-1]$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{extendMult}}$ and receives from the adversary the shares of $[c^{(i)}]_t$ held by corrupted parties. $\mathcal{S}$ also receives the difference $d^{(i)}$ from the adversary.
- For $[c^{(k)}]_t$ and $d^{(k)}$, recall that $[c^{(k)}]_t$ is computed by
$$[c^{(k)}]_t = [z]_t - \sum_{i=1}^{k-1} [c^{(i)}]_t.$$
Therefore, $\mathcal{S}$ can computes the shares held by corrupted parties from the above equation. Furthermore, $\mathcal{S}$ computes $d^{(k)}$ by
$$d^{(k)} = d - \sum_{i=1}^{k-1} d^{(i)}.$$

In Step 4, $\mathcal{S}$ needs to simulate the behaviors of honest parties in Extend-Compress. Note that Extend-Compress only contains local computation and invocations of $\mathcal{F}_{\text{extendMult}}, \mathcal{F}_{\text{coin}}$. For $\mathcal{F}_{\text{coin}}$, $\mathcal{S}$ faithfully generates a random element. For $\mathcal{F}_{\text{extendMult}}$, $\mathcal{S}$ receives from the adversary the shares of corrupted parties and the difference. $\mathcal{S}$ follows Extend-Compress to computes the shares of $[\boldsymbol{a}]_t, [\boldsymbol{b}]_t, [c]_t$ held by corrupted parties and the difference $d' = c - \boldsymbol{a} \odot \boldsymbol{b}$.

– Simulation of Randomization:
Note that, for the input inner-product tuple $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$, the simulator $\mathcal{S}$ learns the shares held by corrupted parties and the difference $d = z - \boldsymbol{x} \odot \boldsymbol{y}$ since they are computed when simulating Dimension-Reduction.
In Randomization, for all $i \in [m]$, $\mathcal{S}$ computes the shares of $[a^{(i)}]_t, [b^{(i)}]_t, [c^{(i)}]_t$ held by corrupted parties and the difference $d^{(i)} = c^{(i)} - a^{(i)} \cdot b^{(i)}$ in the same way as that in Dimension-Reduction. For

$[a^{(0)}]_t, [b^{(0)}]_t$, $\mathcal{S}$ receives from the adversary the shares held by corrupted parties when emulating $\mathcal{F}_{\text{rand}}$. For $[c^{(0)}]_t$, $\mathcal{S}$ receives the shares from the adversary held by corrupted parties and the difference $d^{(0)}$ when emulating $\mathcal{F}_{\text{mult}}$.

In Step 6, $\mathcal{S}$ needs to simulate the behaviors of honest parties in COMPRESS. This can be simulated in the same way as that for EXTEND-COMPRESS. At the end of this step, $\mathcal{S}$ learns the shares of $[a]_t, [b]_t, [c]_t$ held by corrupted parties and also the difference $d' = c - a \cdot b$. $\mathcal{S}$ randomly samples $a, b$ and computes $c = a \cdot b + d'$. Based on the secrets $a, b, c$ and the shares held by corrupted parties, $\mathcal{S}$ reconstructs the whole sharings $[a]_t, [b]_t, [c]_t$.

In Step 7 and Step 8, $\mathcal{S}$ faithfully follows the protocol. Recall that $\mathcal{S}$ receives $b \in \{\texttt{abort}, \texttt{accept}\}$ from $\mathcal{F}_{\text{multVerify}}$ indicating whether the multiplications are correct. If $b = \texttt{accept}$ but an honest party takes `abort` as output, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{multVerify}}$. Otherwise, $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\text{multVerify}}$.

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

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

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

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

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

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

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

However, according to Lemma 8, Lemma 9, Lemma 10, this happens with negligible probability.

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

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

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

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

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

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

field elements in $\mathbb{K}$.

*Remark 5.* We note that the circuit size is bounded by $\text{poly}(\kappa)$ where $\kappa$ 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 6.* Note that MultVerification requires $O(\log_k m)$ rounds. In the real world, one can adjust $k$ based on the overhead of each round and the overhead of sending each bit via a private channel to achieve the best running time.

## 6 Protocol

In this section, we show how to use our new technique to construct a secure-with-abort protocol. Recall that we are in the client-server model where there are $c$ clients and $n = 2t + 1$ servers (denoted by parties). The adversary is able to control up to $c$ clients and $t$ parties. In our construction, the clients only participate in the input phase and the output phase.

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

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

The functionality $\mathcal{F}_{\text{main}}$ is described in Functionality 18. The main protocol Main appears in Protocol 19. 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, $\phi$ is the size of an field element in $\mathbb{F}$ and $\kappa$ is the security parameter.

**Functionality 18:** $\mathcal{F}_{\text{main}}$

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

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

*Proof.* According to Theorem 1, we assume that the adversary controls exactly $t$ parties.

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

*Simulation of* Main*.* We describe the strategy of $\mathcal{S}$ phase by phase.

- Simulation of Input Phase:
  For an input $x$ belongs to an honest client, $\mathcal{S}$ randomly samples $t$ elements as the shares of $[x]_t$ of corrupted parties. Then $\mathcal{S}$ sends these shares to corrupted parties.
  For an input $x$ belongs to a corrupted client, $\mathcal{S}$ receives from the adversary the shares held by honest parties. Note that, $\mathcal{S}$ learns $t+1$ shares of $[x]_t$. $\mathcal{S}$ reconstructs the whole sharing $[x]_t$ and sends $x$ as the input of this corrupted client to $\mathcal{F}_{\text{main}}$.
  Note that for each input sharing, $\mathcal{S}$ learns the shares held by corrupted parties.

**Protocol 19:** Main

1. **Input Phase**:
   For each input $x$ belongs to client, client randomly samples a degree-$t$ sharing $[x]_t$ and distributes the shares to all parties.
2. **Computation Phase**:
   All parties start with holding a degree-$t$ sharing for each input gate. The circuit is evaluated in a predetermined topological order.
   - For each addition gate with input sharings $[x]_t, [y]_t$, all parties locally compute $[z]_t = [x]_t + [y]_t$.
   - For each multiplication gate with input sharings $[x]_t, [y]_t$, all parties invoke $\mathcal{F}_{\text{mult}}$ on $[x]_t, [y]_t$ to compute $[z]_t$ where $z = x \cdot y$.
3. **Verification Phase**:
   All parties invoke $\mathcal{F}_{\text{multVerify}}$ to check the correctness of the multiplications.
4. **Output Phase**:
   For each output gate, let $[x]_t$ be the sharing associated with this gate and client be the client who should receive this output. All parties send their shares of $[x]_t$ to client. client checks whether the shares of $[x]_t$ is consistent. If not, client aborts. Otherwise, client reconstructs the result $x$.

– Simulation of Computation Phase:
  In the computation phase, $\mathcal{S}$ will compute the shares of each sharing held by corrupted parties. Note that this already holds for the sharing of each input gate.
  - For each addition gate with input sharings $[x]_t, [y]_t$, $\mathcal{S}$ computes the shares of $[z]_t = [x]_t + [y]_t$ held by corrupted parties.
  - For each multiplication gate with input sharings $[x]_t, [y]_t$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{mult}}$ and receives the shares of $[z]_t$ held by corrupted parties and the difference $d$.

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

– Simulation of Verification Phase:
  $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{multVerify}}$. Recall that in the simulation of the computation phase, $\mathcal{S}$ has computed the shares of each multiplication tuple held by corrupted parties and the difference. $\mathcal{S}$ directly sends these shares and differences to the adversary as in $\mathcal{F}_{\text{multVerify}}$. If there exists a non-zero difference, $\mathcal{S}$ sets $b = \texttt{abort}$. Otherwise, $\mathcal{S}$ sets $b = \texttt{accept}$. Then $\mathcal{S}$ sends $b$ to the adversary.
  - If $b = \texttt{accept}$ and the adversary replies `continue`, $\mathcal{S}$ moves to the next phase.
  - Otherwise, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{main}}$ and aborts.

– Simulation of Output Phase:
  For each output gate with $[x]_t$ associated with it, if the receiver is an honest client, $\mathcal{S}$ receives from the adversary the shares held by corrupted parties. Then $\mathcal{S}$ checks whether the shares are the same as the ones computed by $\mathcal{S}$. If true, $\mathcal{S}$ accepts the output. Otherwise, $\mathcal{S}$ rejects the output.
  If the receiver is a corrupted client, $\mathcal{S}$ receives the result $x$ from $\mathcal{F}_{\text{main}}$. Then, based on the shares of $[x]_t$ held by corrupted parties and the secret $x$, $\mathcal{S}$ reconstructs the shares held by honest parties, and sends these shares to the adversary.
  Finally, if $\mathcal{S}$ rejects any output of honest clients, $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\text{main}}$. Otherwise, $\mathcal{S}$ sends `continue` to $\mathcal{F}_{\text{main}}$.

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

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

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

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

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

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

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

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

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

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

*Analysis of the Concrete Efficiency.* 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 $O(\log_k C \cdot (kn+n^2)\kappa)$ 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].

## 6.1 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 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 20.

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).

**Lemma 12.** *The protocol* Opt-Mult *securely computes the functionality $\mathcal{F}_{mult}$ in the $\mathcal{F}_{doubleRand}$-hybrid model in the presence of a fully malicious adversary controlling $t$ corrupted parties.*

This lemma can be proved in the same way as that for Lemma 4. Therefore, for simplicity, we omit the details.

**Protocol 20:** Opt-Mult

1. All parties agree on a special party $P_{\mathtt{king}}$. Let $\mathcal{T}$ be a set of $t+1$ parties (including $P_{\mathtt{king}}$) all parties agree on. Let $[x]_t, [y]_t$ denote the input sharings.
2. All parties invoke $\mathcal{F}_{\text{doubleRand}}$ to prepare a pair of random double sharings $([r]_t, [r]_{2t})$.
3. All parties locally compute $[x \cdot y + r]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$.
4. $P_{\mathtt{king}}$ collects all shares and reconstructs the secret value $x \cdot y + r$. Then $P_{\mathtt{king}}$ sets the shares of parties outside of $\mathcal{T}$ to be 0. $P_{\mathtt{king}}$ recovers the whole sharing $[x \cdot y + r]_t$ using these $t$ shares of 0 and the secret value $x \cdot y + r$.
5. $P_{\mathtt{king}}$ 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.
6. All parties locally compute $[x \cdot y]_t = [x \cdot y + r]_t - [r]_t$.

## 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.

BBCG$^+$19. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 67–97, Cham, 2019. Springer International Publishing.

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

BGIN19. Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS ?19, page 869?886, New York, NY, USA, 2019. Association for Computing Machinery.

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

GLOS20. Vipul Goyal, Hanjun Li, Rafail Ostrovsky, and Yifan Song. Fathom: Fast honest majority mpc. Manuscript, 2020.

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 multi-party computation. In *Annual International Cryptology Conference*, pages 101–118. Springer, 2001.

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

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.