Guaranteed Output Delivery Comes Free in Honest Majority MPC

Vipul Goyal[1], Yifan Song[1](✉), and Chenzhi Zhu[2]

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

Abstract. We study the communication complexity of unconditionally secure MPC with guaranteed output delivery over point-to-point channels for corruption threshold $t < n/2$, assuming the existence of a public broadcast channel. We ask the question: "is it possible to construct MPC in this setting s.t. the communication complexity per multiplication gate is linear in the number of parties?" While a number of works have focused on reducing the communication complexity in this setting, the answer to the above question has remained elusive until now. We also focus on the concrete communication complexity of evaluating each multiplication gate.

We resolve the above question in the affirmative by providing an MPC with communication complexity $O(Cn\phi)$ bits (ignoring fixed terms which are independent of the circuit) where ϕ is the length of an element in the field, C is the size of the (arithmetic) circuit, n is the number of parties. This is the first construction where the asymptotic communication complexity matches the best-known semi-honest protocol. This represents a strict improvement over the previously best-known communication complexity of $O(C(n\phi+\kappa)+D_M n^2\kappa)$ bits, where κ is the security parameter and D_M is the multiplicative depth of the circuit. Furthermore, the concrete communication complexity per multiplication gate is 5.5 field elements per party in the best case and 7.5 field elements in the worst case when one or more corrupted parties have been identified. This also roughly matches the best-known semi-honest protocol, which requires 5.5 field elements per gate.

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) with guaranteed output delivery. The adversary is allowed to corrupt at most $t < n/2$ parties and is fully malicious. We assume the existence of private point-to-point communication channels and a public broadcast channel. We are interested in the communication complexity of the secure MPC, which is measured by the number of bits X via private point-to-point channels and the number of bits Y via the public broadcast

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.

C. Zhu—Work done in part while at CMU.

channel, i.e., $X + Y \cdot \mathcal{BC}$. The first positive solutions in this setting were proposed in [RBO89,Bea89]. After those, several subsequent works [CDD+99,BTH06,BSFO12] have focused on improving the communication complexity of the protocol. 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.

In this paper, we ask the following natural question:

"Is it possible to construct unconditional MPC with guaranteed output delivery for $t < n/2$ s.t. the communication complexity per multiplication gate is linear in the number of parties? Furthermore, what is the concrete communication complexity per multiplication gate?"

Having linear communication complexity per multiplication gate greatly benefits the scalability of the protocol, as it means that the work done by each party is independent of the number of parties but only related to the size of the circuit. While a number of works have made significant progress, this question has remained opened until now.

The best-known result in this setting is the construction in the work [BSFO12] of Ben-Sasson, Fehr and Ostrovsky. The construction in [BSFO12] has communication complexity $O(C(n\phi + \kappa) + D_M n^2 \kappa)$ bits (ignoring fixed terms which are independent of the circuit), where C is the size of the circuit, ϕ is the length of a field element, κ is the security parameter and D_M is the multiplicative depth of the circuit. Comparing with the best-known result against semi-honest adversaries in [DN07], which has communication complexity $O(Cn\phi)$ bits, there is an additional term $D_M n^2 \kappa$ related to the circuit. In the worst case where the circuit is "narrow and deep", $D_M n^2 \kappa$ may even become the dominating term of the communication complexity and result in $O(n^2)$ elements per gate. Ben-Sasson et. al asked if this quadratic term related to the depth of the circuit is inherent.

In a beautiful work, Ishai et al. [IKP+16] provided a general transformation from a protocol in the setting of security with abort to a protocol with guaranteed output delivery. Instantiation this transformation with the best-known protocol for security with abort, the resulting construction eliminates the quadratic term w.r.t. the circuit depth. However, the communication complexity of the resulting protocol now has a term $O(W \cdot \text{poly}(n))$, where W is the width of the circuit, and, $\text{poly}(n)$ can be at least n^4 for certain circuits. For the circuit with a large width, this term may even become the dominating term.

In the setting of $t < n/3$ corruptions (where a public broadcast channel can be securely simulated), question of getting a construction with linear communication complexity was recently resolved in the recent work of Goyal et. al [GLS19], which presented a construction with communication complexity $O(Cn\phi)$ bits. Similar results were also known in the setting of security with abort in [GIP+14,LN17,CGH+18,NV18,GS20].

Our Results. In this work, we answer the above question in the affirmative by presenting an MPC protocol with communication complexity $O(Cn\phi)$ bits (ignoring fixed terms which are independent of the circuit). Furthermore, we also focus on the concrete efficiency, i.e., the number of elements per multiplication gate per party. Concretely, our result achieves $5.5 + \epsilon$ elements in the best case and $7.5 + \epsilon$ elements in the worst case when one or more corrupted parties have been identified, where ϵ can be an arbitrarily small constant. Comparing with the best-known result [DN07] in the semi-honest setting, which requires 6 elements, and the best-known result [GS20] in the setting of security with abort, which requires 5.5 elements, our result essentially shows that achieving output delivery guarantee requires no additional cost compared to semi-honest security and malicious security (with abort).

Our main contributions lie in two aspects, (1) we present the first construction in this setting where the asymptotic communication complexity matches that in the semi-honest setting, and, (2) our protocol roughly achieves the same concrete efficiency as the best-known semi-honest protocol. These improvements stem from the idea of developing a suite of techniques to efficiently compile the best-known secure-with-abort protocol [GS20] into a fully secure protocol. Additionally, we introduce a technique which allows us to re-use authentication keys towards developing a more efficient verifiable secret sharing scheme. An overview of our new ideas can be found in Section 2.

Related Works. In this section, we compare our result with several related constructions in both techniques and efficiency. In the following, let C denote the size of the circuit, ϕ denote the size of a field element, κ denote the security parameter, D_M denote the depth of the circuit, and W denote the width of the circuit. We will ignore fixed terms which are independent of the circuit.

Comparison with [BSFO12]. The construction in [BSFO12] is most related to our result. In fact, we reuse and modify many protocols in [BSFO12] in our construction.

The communication complexity achieved by the construction in [BSFO12] is $O(C(n\phi+\kappa)+D_M n^2\kappa)$ bits. Our result removes both the quadratic term related to D_M and the term $O(C\kappa)$. Furthermore, the use of Beaver triples for multiplication gates in [BSFO12] is more expensive than the multiplication protocol in the best-known secure-with-abort protocol [GS20]. As a result, the communication cost per multiplication gate in [BSFO12] is a fixed 20 field elements (without considering the effect of $O(D_M n^2\kappa)$). Our result achieves $5.5+\epsilon$ field elements per multiplication gate in the best case and $7.5+\epsilon$ field elements in the worst case when one or more corrupted parties have been identified, where ϵ can be an arbitrarily small constant. In the best case, our result matches the best-known semi-honest protocol [DN07] and the best-known secure-with-abort protocol [GS20].

Technically, while the construction from [BSFO12] uses Beaver triples to compute multiplications in the computation phase, we directly use a modified version of the multiplication protocol of the best-known protocol [DN07] from the semi-honest setting. We note that Beaver triples provide plenty of redundancy which simplifies the checking process in the computation phase. However, the use of Beaver triples unfortunately requires a verification for each layer of the circuit, which leads to the quadratic term related to D_M. On the other hand, we start from the best-known secure-with-abort protocol [GS20], which does not make use of Beaver triples. While this idea can potentially remove the term $O(D_M n^2)$, without the redundancy provided by Beaver triples, the verification becomes difficult and even the computation cannot proceed when malicious parties refuse to participate in the computation. We will show how to tackle these difficulties in Section 2.

Comparison with [IKP+16]. Ishai et al. [IKP+16] provided a general transformation from a protocol in the setting of security with abort to a protocol with guaranteed output delivery. When instantiating their transformation with the best-known protocols [GS20] in the setting of security with abort, the resulting protocol can achieve 5.5 field elements per multiplication gate when the *width* of the circuit is small.

However, a drawback of this transformation is that the efficiency of the resulting protocol has a large dependency on the width of the circuit. Specifically, the communication complexity of the resulting protocol contains a term $O(W \cdot \mathrm{poly}(n,\kappa))$ (where poly is relatively large). For the circuit with a large width, this term may even become the dominating term.

Comparison with [GLS19]. Recently, Goyal et al. [GLS19] gave the first construction against $1/3$ corruption such that the communication complexity per multiplication gate is linear in the number of parties. The communication complexity is $O(Cn\phi)$ bits. Since they mainly focused on the feasibility and the protocol is perfectly secure, the concrete efficiency is 66 elements per multiplication gate.

Unfortunately the techniques developed in [GLS19] fail in the setting of honest majority. Technically, we use a significantly different approach from that in [GLS19] to remove the quadratic term related to the circuit depth. The reason for $O(D_M n^2)$ is that all parties need to ensure the correctness of multiplications in one layer before moving on to the next layer. To this end, each layer requires at least $O(n^2)$ communication, which results in $O(D_M n^2)$ overhead. While Goyal et al. [GLS19] used n-out-of-n secret sharings to overcome the layer restriction, our approach is to directly compile the best-known secure-with-abort protocol [GS20], which does not have the term $O(D_M n^2)$, to a fully secure one.

Comparison with [GS20]. The recent work [GS20] in the setting of security with abort shows that the concrete efficiency can be the same as the best-known semi-honest protocol [DN07]. Specifically, the protocol [GS20] achieves the asymptotic complexity $O(Cn\phi)$ bits and concrete efficiency of 5.5 field elements per multiplication gate per party. In the best case, our protocol matches the concrete efficiency of [GS20].

Technically, we directly compile the protocol [GS20] into a fully secure one. While a secure-with-abort protocol simply aborts when a failure occurs in the computation, we need to find out where things went wrong and ensure the success of the computation. However, due to the lack of redundancy (compared with the protocol [BSFO12] which uses Beaver triples), the verification becomes difficult and even the computation cannot proceed when malicious parties refuse to participate in the computation. We address these two problems in Section 2.

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

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 where $t < n/3$, 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., Damgard et al. [DIK10] introduced a protocol with communication complexity $O(C \log C \log n \cdot \kappa + D_M^2 \text{poly}(n, \log C)\kappa)$ bits.

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

2 Technical Overview

Our 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.

In the following, we will use a variable with bold font $\boldsymbol{x}$ to represent a vector.

2.1 Background: Using An Efficient Secure-With-Abort Protocol without $O(D_M n^2)$

We observe that several recent secure-with-abort protocols [GIP+14,CGH+18,NV18,GS20] in this setting do not have the factor $O(D_M n^2)$ in the communication complexity. In particular, the best-known result [GS20] has achieved asymptotic communication complexity $O(Cn\phi)$ bits (ignoring fixed terms which are independent of the circuit) and concrete efficiency of 5.5 field elements per gate, which matches the best-known result in the semi-honest setting [DN07].

Therefore, our starting idea is to compile the protocol in [GS20] into one with guaranteed output delivery. Hopefully, it will help us remove the factor $O(D_M n^2)$ and achieve the same concrete efficiency as the semi-honest setting. We first give a sketch of the construction from [GS20].

Overview. The protocol in [GS20] is composed of the best-known semi-honest protocol [DN07], which we referred to as the DN protocol, and an efficient verification for a batch of multiplication gates.

The high-level idea of the DN protocol is to let all parties compute a degree-t Shamir secret sharing for each wire. For an addition gate, the output sharing can be computed by locally adding the input sharings. For a multiplication gate, we directly describe an extension of the DN multiplication protocol [GS20] (which is a slightly optimized version of that in [CGH+18]), which allows all parties to compute an inner-product of two vectors of sharings $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t$. The original DN multiplication protocol is a special case when the dimension is 1.

For each inner-product operation, all parties will prepare a pair of random sharings $([r]_t, [r]_{2t})$, which we refer to as double sharings. In brief, double sharings are prepared in the following manner:

1. Each party generates and distributes a pair of random double sharings.
2. Each pair of double sharings is a linear combination of the random double sharings distributed by each party.

Note that a random degree-t sharing can be prepared in the same way except that all parties only distribute the degree-t sharing. Then, all parties execute the following steps to compute $[\boldsymbol{x} \odot \boldsymbol{y}]_t$, where $\odot$ denotes the inner-product operation.

1. All parties first locally compute $[e]_{2t} := [\boldsymbol{x}]_t \odot [\boldsymbol{y}]_t + [r]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[e]_{2t}$ and reconstructs the secret e. Then $P_{\texttt{king}}$ generates a degree-t sharing $[e]_t$ and distributes the shares to all other parties.
3. All parties locally compute $[\boldsymbol{x} \odot \boldsymbol{y}]_t = [e]_t - [r]_t$.

Here $P_{\texttt{king}}$ is the party all parties agree on in the beginning. We point out that the communication cost is independent of the dimension of the input vectors.

Note that the DN protocol only provides security against semi-honest adversaries. After evaluating the whole circuit, all parties together verify the multiplications. If the check passes, all parties proceed to reconstruct the output. Otherwise, the protocol aborts.

Sketch of Batch-wise Multiplication Verification in [GS20]. 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 verification contains three steps, (1) De-Linearization, (2) Dimension-Reduction, and (3) Recursion and Randomization.

Step One: De-Linearization. The first step is transforming the check of m multiplication tuples into a check of an inner-product tuple of dimension m. Note that simply setting $[\boldsymbol{x}]_t = ([x^{(1)}]_t, \ldots, [x^{(m)}]_t), [\boldsymbol{y}]_t = ([y^{(1)}]_t, \ldots, [y^{(m)}]_t), [z]_t = \sum_{i=1}^{m} [z^{(i)}]_t$ and checking the correctness of $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$ is insufficient. For example, the corrupted parties may only cause $z^{(1)} = x^{(1)} \cdot y^{(1)} + 1$ and $z^{(2)} = x^{(2)} \cdot y^{(2)} - 1$. We cannot detect it by using this approach. Therefore, we need to add some randomness so that the inner-product tuple will be incorrect with overwhelming probability if any one of the original multiplication tuples is incorrect.

Consider the following two polynomials:

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

Note that if some multiplication tuple is incorrect, we will have $F \neq G$. In this case, the number of λ such that $F(\lambda) = G(\lambda)$ is bounded by $m - 1$. Therefore, for a random λ, $F(\lambda) \neq G(\lambda)$ with overwhelming probability.

All parties will generate a random element λ as challenge. Then, the inner-product tuple $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$ is set to be

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

Note that $F(\lambda) = \boldsymbol{x} \odot \boldsymbol{y}$ and $G(\lambda) = z$. Therefore, 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 a single inner-product tuple, 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, the second step is to reduce the dimension of the inner-product tuple we want to verify. It is achieved by using a natural extension of the Batch-wise Multiplication Verification technique in [BSFO12]. In short, this technique can compresses the check of ℓ multiplication tuples into one check of a new generated multiplication tuple. However, the main drawback of this technique is that it requires additional ℓ multiplication operations to do the compression.

A natural extension of Batch-wise Multiplication Verification [NV18] is to compress the check of ℓ *inner-product* tuples into one check of a new generated *inner-product* tuple. The communication cost is roughly ℓ inner-product operations, which is the same as ℓ multiplication operations.

Let k be a compression parameter. The goal is to transform the original inner-product tuple of dimension m to be a new inner-product tuple of dimension m/k. The two input vectors of sharings are first chopped into k equal parts:

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

where $\{\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\}_{i\in[k]}$ are vectors of dimension m/k. For each $i \in [k-1]$, all parties compute the inner-product of $([\boldsymbol{x}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t)$ using the extension of the DN multiplication protocol. Let $[z^{(i)}]_t$ denote the result. Then set $[z^{(k)}]_t = [z]_t - \sum_{i=1}^{k-1}[z^{(i)}]_t$ to be the result for the inner-product of $([\boldsymbol{x}^{(k)}]_t, [\boldsymbol{y}^{(k)}]_t)$. In this way, if the original inner-product tuple is incorrect, then at least one of the new inner-product tuples is incorrect.

Finally, all parties use the extension of Batch-wise Multiplication Verification [NV18] to compress the check of these k inner-product tuples into one check of a single inner-product tuple. In particular, the output inner-product tuple has dimension m/k.

Step Three: Recursion and Randomization. The second step is repeated $\log_k m$ times so that all parties only need to check the correctness of *a single* multiplication tuple in the end. To simplify the checking process for the last tuple, all parties will prepare a random multiplication tuple as a random mask in the last call of the second step.

Concretely, in the last call of the second step, all parties need to compress the check of k multiplication tuples into one check of a single multiplication tuple. A random multiplication tuple will be included as a random mask of these k multiplication tuples in the compression. That is, the last call will compress the check of $k+1$ multiplication tuples into one check of a single multiplication tuple, which we refer to as the *ultimate multiplication tuple*. In this way, to check the ultimate 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 DN protocol.
2. All parties compute $[c]_t = [a \cdot b]_t$ using the DN multiplication protocol.

Problems with the Starting Idea. The most direct problem of using the secure-with-abort protocol in [GS20] is that a single error leads to an abortion of the whole computation. However, our purpose is to build a protocol with guaranteed output delivery, which should ensure the success of the computation no matter how corrupted parties behave. It means that, when facing a failure in the check of the ultimate multiplication tuple, we need to find out where things went wrong and be able to proceed the computation.

Another problem is that, when a corrupted party maliciously refuses to participate in the computation or an identified corrupted party is kicked out from the computation, the DN protocol cannot even proceed. This is because in the DN multiplication protocol, $P_{\texttt{king}}$ needs to reconstruct a degree-$2t$ sharing $[e]_{2t} := [\boldsymbol{x}]_t \odot [\boldsymbol{y}]_t + [r]_{2t}$. $P_{\texttt{king}}$ needs $2t+1 = n$ shares to do reconstruction. This cannot be achieved if some party does not send its share to $P_{\texttt{king}}$.

In the following, we will tackle these two problems respectively.

2.2 Efficient Verification Using Virtual Transcripts

To be able to identify the corrupted parties that deviate from the protocol when a failure occurs in the check of the ultimate multiplication tuple, our idea is to compute a *virtual transcript* of the ultimate multiplication tuple. A virtual transcript can be seen as the transcript where all parties directly compute the ultimate multiplication tuple using the DN multiplication protocol. Although the transcript does not correspond to a real execution, all parties should agree on the messages they sent in a virtual transcript. In the case that a failure occurs in the check of the ultimate multiplication tuple, all parties can open the whole virtual transcripts to identify the parties which behaved maliciously.

We first give a sketch of the extension of Batch-wise Multiplication Verification [NV18].

Extension of Batch-wise Multiplication Verification [NV18]. Suppose we have ℓ inner-product tuples $\{([\boldsymbol{x}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^{\ell}$ and would like to verify whether $z^{(i)} = \boldsymbol{x}^{(i)} \odot \boldsymbol{y}^{(i)}$ for all $i \in [\ell]$. The extension of Batch-wise Multiplication Verification [NV18] works as follows.

1. Let $\boldsymbol{F}(\cdot), \boldsymbol{G}(\cdot)$ be two vectors of degree-$(\ell-1)$ polynomials such that

   $$\forall i \in [\ell], \quad \boldsymbol{F}(i) = \boldsymbol{x}^{(i)},\ \boldsymbol{G}(i) = \boldsymbol{y}^{(i)}.$$

   All parties can locally compute the shares of $[\boldsymbol{F}(\cdot)]_t$ and $[\boldsymbol{G}(\cdot)]_t$ by using their shares of $[\boldsymbol{x}^{(1)}]_t, \ldots, [\boldsymbol{x}^{(\ell)}]_t$ and $[\boldsymbol{y}^{(1)}]_t, \ldots, [\boldsymbol{y}^{(\ell)}]_t$, i.e., by doing interpolation on their own vectors of shares.
2. All parties compute $[\boldsymbol{x}^{(i)}]_t = [\boldsymbol{F}(i)]_t, [\boldsymbol{y}^{(i)}]_t = [\boldsymbol{G}(i)]_t$ for all $i \in \{\ell+1, \ldots, 2\ell-1\}$.
3. For all $i \in \{\ell+1, \ldots, 2\ell-1\}$, all parties compute $[z^{(i)}]_t$ where $z^{(i)} = \boldsymbol{x}^{(i)} \odot \boldsymbol{y}^{(i)}$ using the extension of the DN multiplication protocol.
4. Let $H(\cdot)$ be a degree-$2(\ell-1)$ polynomial such that

   $$\forall i \in [2\ell-1], \quad H(i) = z^{(i)}.$$

   All parties can locally compute the shares of $[H(\cdot)]_t$ by using their shares of $[z^{(1)}]_t, \ldots, [z^{(2\ell-1)}]_t$, i.e., by doing interpolation on their own shares.

Note that if all inner-product tuples $\{([\boldsymbol{x}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^{2\ell-1}$ are correct, we should have $\boldsymbol{F} \odot \boldsymbol{G} = H$. Otherwise, $\boldsymbol{F} \odot \boldsymbol{G} \neq H$, and the number of λ such that $\boldsymbol{F}(\lambda) \odot \boldsymbol{G}(\lambda) = H(\lambda)$ is bounded by $2(\ell-1)$. Therefore, to verify the original ℓ inner-product tuples, it is sufficient to sample a random point λ and only verify $([\boldsymbol{F}(\lambda)]_t, [\boldsymbol{G}(\lambda)]_t, [H(\lambda)]_t)$. We refer to $([\boldsymbol{F}(\lambda)]_t, [\boldsymbol{G}(\lambda)]_t, [H(\lambda)]_t)$ as the final inner-product tuple.

Preparing Virtual Transcript for the Final Inner-product Tuple. We note that the transcript of the extension of the DN multiplication protocol contains 7 sharings

$$([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t).$$

The idea of the virtual transcript is to recover the missing parts $[r]_t, [r]_{2t}, [e]_{2t}, [e]_t$. Therefore, in the case that the check of the final inner-product tuple fails, by examining the corresponding virtual transcripts, we can find out where things went wrong and potentially identify a corrupted party.

Recall that the final inner-product tuple $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$ is derived by using polynomial interpolation on $2\ell-1$ inner-product tuples. In a similar way, we derive $[r]_t, [r]_{2t}, [e]_{2t}, [e]_t$ by polynomial interpolation on the corresponding values in the transcripts of these $2\ell-1$ inner-product tuples.

In more detail, given the transcripts of the original m inner-product tuples

$$\{([\boldsymbol{x}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^{\ell},$$

we want to compute the transcript of the resulting tuple.

Let $\{([\boldsymbol{x}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)\}_{i=\ell+1}^{2\ell-1}$ denote the transcripts generated in the extension of Batch-wise Multiplication Verification. Recall that $[\boldsymbol{F}(\cdot)]_t, [\boldsymbol{G}(\cdot)]_t, [H(\cdot)]_t$ satisfy that

$$\forall i \in [2\ell-1]: \ [\boldsymbol{F}(i)]_t = [\boldsymbol{x}^{(i)}]_t,\ [\boldsymbol{G}(i)]_t = [\boldsymbol{y}^{(i)}]_t,\ [H(i)]_t = [z^{(i)}]_t.$$

Let $[R(\cdot)]_t, [R(\cdot)]_{2t}, [E(\cdot)]_{2t}, [E(\cdot)]_t$ be sharings of polynomials of degree $2(m-1)$ such that

$$\begin{aligned}\forall i \in [2\ell-1]: \quad & [R(i)]_t = [r^{(i)}]_t,\ [R(i)]_{2t} = [r^{(i)}]_{2t}, \\ & [E(i)]_{2t} = [e^{(i)}]_{2t},\ [E(i)]_t = [e^{(i)}]_t.\end{aligned}$$

Therefore, we have $[E(\cdot)]_{2t} = [\boldsymbol{F}(\cdot)]_t \odot [\boldsymbol{G}(\cdot)]_t + [R(\cdot)]_{2t}$ and $[H(\cdot)]_t = [E(\cdot)]_t - [R(\cdot)]_t$. It means that, for every λ, one can regard

$$([\boldsymbol{F}(\lambda)]_t, [\boldsymbol{G}(\lambda)]_t, [R(\lambda)]_t, [R(\lambda)]_{2t}, [E(\lambda)]_{2t}, [E(\lambda)]_t, [H(\lambda)]_t)$$

as a transcript of the following steps:

1. All parties first locally compute $[E(\lambda)]_{2t} := [\boldsymbol{F}(\lambda)]_t \odot [\boldsymbol{G}(\lambda)]_t + [R(\lambda)]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[E(\lambda)]_{2t}$ and reconstructs the secret $E(\lambda)$. Then $P_{\texttt{king}}$ generates a degree-t sharing $[E(\lambda)]_t$ and distributes the shares to all other parties.
3. All parties locally compute $[H(\lambda)]_t = [E(\lambda)]_t - [R(\lambda)]_t$.

To this end, all parties locally compute the shares of $[R(\cdot)]_t, [R(\cdot)]_{2t}$ by using their shares of $[r^{(1)}]_t, \ldots, [r^{(2\ell-1)}]_t$ and $[r^{(1)}]_{2t}, \ldots, [r^{(2\ell-1)}]_{2t}$. Then set $[E(\cdot)]_{2t} = [\boldsymbol{F}(\cdot)]_t \odot [\boldsymbol{G}(\cdot)]_t + [R(\cdot)]_{2t}$ and $[E(\cdot)]_t = [H(\cdot)]_t + [R(\cdot)]_t$. $P_{\texttt{king}}$ further computes $[E(\cdot)]_{2t}$ by using the sharings $[e^{(1)}]_{2t}, \ldots, [e^{(2m-1)}]_{2t}$ it received, and $[E(\cdot)]_t$ by using the sharings $[e^{(1)}]_t, \ldots, [e^{(2m-1)}]_t$ it distributed.

All parties generate a random element λ as challenge. The transcript

$$([\boldsymbol{F}(\lambda)]_t, [\boldsymbol{G}(\lambda)]_t, [R(\lambda)]_t, [R(\lambda)]_{2t}, [E(\lambda)]_{2t}, [E(\lambda)]_t, [H(\lambda)]_t)$$

is what we want to verify.

Preparing Virtual Transcript for the Ultimate Multiplication Tuple. We will follow the Batch-wise Multiplication Verification in [GS20] and prepare a virtual transcript for the tuple generated in each step. Suppose the transcripts of the original m multiplication tuples are

$$\{([x^{(i)}]_t, [y^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^m,$$

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

Step One: De-Linearization. Recall that in Step One, all parties first generate a random element λ and set

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

The virtual transcript for $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [z]_t)$ can be prepared by setting

$$([r]_t, [r]_{2t}, [e]_{2t}, [e]_t) = \sum_{i=1}^m \lambda^{i-1}([r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t).$$

The transcript $([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t)$ is what we need to verify. Note that this transcript corresponds to a single inner-product tuple of dimension m.

Step Two: Dimension-Reduction. Recall that in Step Two, we want to reduce the dimension of the inner-product tuple from Step One. Let

$$([\boldsymbol{x}]_t, [\boldsymbol{y}]_t, [r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t)$$

denote the transcript. Recall that $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t$ are first chopped into k equal parts:

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

where $\{\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\}_{i\in[k]}$ are vectors of dimension m/k. For each $i \in [k-1]$, all parties compute $[z^{(i)}]_t = [\boldsymbol{x}^{(i)} \odot \boldsymbol{y}^{(i)}]_t$ by using the extension of the DN multiplication protocol. Let $([r^{(i)}]_t, [r^{(i)}]_{2t})$ be the corresponding double sharings used by the parties, $[e^i]_{2t}, [e^{(i)}]_t$ be the sharings which $P_{\texttt{king}}$ received and sent respectively. Hence,

$$([\boldsymbol{x}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)$$

denote the transcript for the inner-product tuple $([\boldsymbol{x}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t, [z^{(i)}]_t)$. So far, we have only used $[\boldsymbol{x}]_t, [\boldsymbol{y}]_t$ from the input inner-product tuple. To ensure that if the input transcript of the inner-product tuple is incorrect, then one of the new generated transcripts is also incorrect, the transcript of the last tuple is computed from the input transcript. By setting

$$\begin{aligned}&([r^{(k)}]_t, [r^{(k)}]_{2t}, [e^{(k)}]_{2t}, [e^{(k)}]_t, [z^{(k)}]_t)\\ =\ &([r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t) - \sum_{i=1}^{k-1}([r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t),\end{aligned}$$

the transcript for $([\boldsymbol{x}^{(k)}]_t, [\boldsymbol{y}^{(k)}]_t)$ is

$$([\boldsymbol{x}^{(k)}]_t, [\boldsymbol{y}^{(k)}]_t, [r^{(k)}]_t, [r^{(k)}]_{2t}, [e^{(k)}]_{2t}, [e^{(k)}]_t, [z^{(k)}]_t).$$

Now we can use the extension of Batch-wise Multiplication Verification [NV18] to compress these k transcripts of inner-product tuples into one transcript of a single inner-product tuple as we described above.

Step Three: Recursion and Randomization. In this step, all parties first recursively invoke Step Two to reduce the dimension of the inner-product tuple from m to k. In the meantime, all parties will also recursively prepare the virtual transcripts.

All parties then prepare a random multiplication tuple, and include this tuple when doing the last call of compression. After all parties prepare this random multiplication tuple and its transcript, all parties can do the same way as that in Step Two to get a transcript of a single multiplication tuple. Let

$$([x^\star]_t, [y^\star]_t, [r^\star]_t, [r^\star]_{2t}, [e^\star]_{2t}, [e^\star]_t, [z^\star]_t)$$

denote the transcript for the ultimate multiplication tuple. It can be regarded as the transcript where all parties run the following steps:

1. All parties first locally compute $[e^\star]_{2t} := [x^\star]_t \cdot [y^\star]_t + [r^\star]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[e^\star]_{2t}$ and reconstructs the secret $e^\star$. Then $P_{\texttt{king}}$ generates a degree-t sharing $[e^\star]_t$ and distributes the shares to all other parties.
3. All parties locally compute $[z^\star]_t = [e^\star]_t - [r^\star]_t$.

Checking the Virtual Transcript. Recall that all parties have opened $[x^\star]_t$, $[y^\star]_t, [z^\star]_t$ to verify the ultimate multiplication tuple. In the case that $([x^\star]_t, [y^\star]_t, [z^\star]_t)$ is not a correct multiplication tuple, all parties will publish their shares of $[r^\star]_t, [r^\star]_{2t}$, $[e^\star]_{2t}, [e^\star]_t$. In addition, $P_{\texttt{king}}$ will publish the whole sharing $[e^\star]_{2t}$ it received and the whole sharing $[e^\star]_t$ it distributed. Then all parties must observe one of the following cases:

– The input sharings $[x^\star]_t, [y^\star]_t$ are inconsistent.
– The pair of double sharings $([r^\star]_t, [r^\star]_{2t})$ is incorrect or inconsistent.
– Some party P_i does not follow the protocol.
– Two parties $(P_i, P_{\texttt{king}})$ do not agree on the message sent from one party to the other party.

For the first two cases, there will be another protocol to help find errors. The main observation is that each sharing $[x]_t$ can be decomposed into $[x]_t = \sum_{i=1}^{n} [x(i)]_t$ where $[x(i)]_t$ is a linear combination of the sharings dealt by P_i. In other words, P_i should be responsible for the consistency of $[x(i)]_t$. Therefore, all parties will check each $[x(i)]_t$ to find errors.

For the last two cases, we can immediately identify a corrupted party or a pair of parties which have conflict with each other. We refer to this pair of parties as a pair of *disputed parties*.

In summary, all parties will finally identify either a corrupted party or a pair of disputed parties.

2.3 Relying on a Small Surgery to Proceed

Now suppose a corrupted party causes the computation to fail and has been identified using the described checks. What do we do? A straightforward idea is to restart the whole computation with the corrupted party excluded and a smaller corruption threshold. In the worst case, however, we may need to rerun the whole protocol $O(n)$ times, which is too expensive. To reduce the penalty due to failures, we rely on Dispute Control [BTH06], which is a general strategy to achieve unconditional security efficiently.

At a high-level, the whole circuit will be partitioned into several small segments. These segments will be evaluated in sequence. In the case that a failure occurs, the computation of this segment is discarded and all parties restart to evaluate the current segment. In other words, the end of each segment is served as a checkpoint. However, one problem with this strategy is that we cannot easily restart the computation with a smaller corruption threshold. This is because all the input sharings, which come from the end of last segment, are shared using the threshold t. Changing threshold means that one need to re-share all the input sharings. In fact, it is the main reason of the factor of $O(W \cdot \mathsf{poly}(n))$ in [IKP$^+$16], where W is the width of the circuit.

To avoid the expensive re-sharing process, we would like to keep the corruption threshold unchanged. Furthermore, we also want to keep the influence on the concrete efficiency as little as possible. To be able to let the protocol proceed without changing the corruption threshold, our idea is to prepare the shares held by identified corrupted parties so that $P_{\texttt{king}}$ will have enough shares to reconstruct a degree-$2t$ sharing.

Notation. Recall that n is the number of all parties and t is the number of corrupted parties. We have $n = 2t + 1$. Let $\mathcal{P}$ be the set of all parties, $\mathcal{C}orr$ be the set of parties which have been identified as corrupted parties so far, and $\mathcal{P}_{\texttt{active}} = \mathcal{P} \backslash \mathcal{C}orr$ be the set of remaining parties. If a party is identified as a corrupted party, it will not participate in the rest of the computations. Hereafter, we use *all parties* to refer parties in $\mathcal{P}_{\texttt{active}}$.

Overview. Recall that for each multiplication gate with input sharings $([x]_t, [y]_t)$, all parties first prepare a pair of random double sharings $([r]_t, [r]_{2t})$. Then all parties execute the following steps to compute $[x \cdot y]_t$.

1. All parties first locally compute $[e]_{2t} := [x]_t \cdot [y]_t + [r]_{2t}$.
2. $P_{\texttt{king}}$ collects all shares of $[e]_{2t}$ and reconstructs the secret e. Then $P_{\texttt{king}}$ generates a degree-t sharing $[e]_t$ and distributes the shares to all other parties.
3. All parties locally compute $[x \cdot y]_t = [e]_t - [r]_t$.

In our construction, when a party P_d needs to generate a random sharing, we require that the shares held by parties in $\mathcal{C}orr$ should be 0. Note that, it does not break the secrecy of the random sharing since parties in $\mathcal{C}orr$ are corrupted. We observe the following two facts.

1. During the generation process of $([r]_t, [r]_{2t})$, each dealer sets the shares held by parties in $\mathcal{C}orr$ to be 0. Since $([r]_t, [r]_{2t})$ is a linear combination of the double sharings dealt by each party, the shares of $([r]_t, [r]_{2t})$ held by parties in $\mathcal{C}orr$ are all 0.
2. For each party P_i, if the i-th share of either $[x]_t$ or $[y]_t$ is 0, then the i-th share of $[x \cdot y]_{2t} := [x]_t \cdot [y]_t$ is also 0.

Our idea is doing a small "surgery" to one input sharing $[x]_t$. Roughly speaking, this means changing the shares of $[x]_t$ held by parties in $\mathcal{C}orr$ to 0 while keeping the secret value x. Let $[\tilde{x}]_t$ denote the sharing after the "surgery". Then, it satisfies that $\tilde{x} = x$ and the shares of $[\tilde{x}]_t$ held by parties in $\mathcal{C}orr$ are 0. Detailed procedure for this "surgery" will be introduced at a later point.

Recall that the shares of $[\tilde{x}]_t, [r]_{2t}$ held by parties in $\mathcal{C}orr$ are 0. Now, when we invoke the DN multiplication protocol on $([\tilde{x}]_t, [y]_t)$, the shares of $[e]_{2t} := [\tilde{x}]_t \cdot [y]_t + [r]_{2t}$ held by parties in $\mathcal{C}orr$ are also 0. Therefore, $P_{\texttt{king}}$ can reconstruct $[e]_{2t}$ by setting the shares held by parties in $\mathcal{C}orr$ to be 0. Thus, each multiplication can be evaluated in two steps, (1) doing a small "surgery" to $[x]_t$, and (2) invoking the DN multiplication protocol on $([\tilde{x}]_t, [y]_t)$. We refer to the first step as Refresh and the second step as PartialMult.

Refresh*: Performing the "Surgery".* Since parties in $\mathcal{C}orr$ are all corrupted, there is no need to protect the secrecy of their shares. The high-level idea is letting $P_{\texttt{king}}$ learn the shares of $[x]_t$ held by parties in $\mathcal{C}orr$. Then $P_{\texttt{king}}$ distributes a random degree-t sharing $[o]_t$ such that $o = 0$ and the shares of $[o]_t, [x]_t$ held by parties in $\mathcal{C}orr$ are the same. Therefore $[\tilde{x}]_t := [x]_t - [o]_t$ is what we need.

In more detail, all parties first prepare a random degree-t sharing $[r]_t$ (as that in the DN protocol). Recall that, in the generation process of $[r]_t$, each dealer sets the shares of parties in $\mathcal{C}orr$ to be 0. Therefore, the shares of $[r]_t$ held by parties in $\mathcal{C}orr$ are 0. Then, all parties run the following steps.

1. All parties locally compute $[e]_t := [x]_t + [r]_t$. Note that the shares of $[e]_t, [x]_t$ held by parties in $\mathcal{C}orr$ are the same.
2. $P_{\texttt{king}}$ collects all shares of $[e]_t$ and computes the shares held by parties in $\mathcal{C}orr$.
3. $P_{\texttt{king}}$ generates and distributes a random degree-t sharing $[o]_t$ where $o = 0$ and the shares of $[o]_t, [e]_t$ held by parties in $\mathcal{C}orr$ are the same.
4. All parties set $[\tilde{x}]_t := [x]_t - [o]_t$.

PartialMult*: Multiplying* $[\tilde{x}]_t$ *and* $[y]_t$*.* To compute $[z]_t$, all parties invoke the multiplication protocol in [DN07] on $([\tilde{x}]_t, [y]_t)$. All parties first prepare a pair of double sharings $([r]_t, [r]_{2t})$ (as that in the DN protocol). Recall that, the shares of $[r]_t, [r]_{2t}$ held by parties in $\mathcal{C}orr$ are 0. Then, all parties run the following steps.

1. All parties locally compute $[e]_{2t} := [\tilde{x}]_t \cdot [y]_t + [r]_{2t}$.
2. $P_{\texttt{king}}$ collects shares of $[e]_{2t}$ from parties in $\mathcal{P}_{\texttt{active}}$. For each party $P_i \in \mathcal{C}orr$, $P_{\texttt{king}}$ sets the i-th share of $[e]_{2t}$ to be 0. Then $P_{\texttt{king}}$ generates a degree-t sharing $[e]_t$ and distributes the shares to all other parties.
3. All parties locally compute $[z]_t = [e]_t - [r]_t$.

Reducing the Communication of Refresh and PartialMult. We note that, to reconstruct a degree-t sharing, $P_{\texttt{king}}$ only needs $t + 1$ shares. Therefore, there is no need to let all parties receive the shares of $[r]_t$. In the beginning of each segment, all parties agree on a set of parties $\mathcal{T} \subseteq \mathcal{P}_{\texttt{active}}$ such that (1) $|\mathcal{T}| = t + 1$, and (2) $P_{\texttt{king}} \in \mathcal{T}$. In brief, $\mathcal{T}$ contains $P_{\texttt{king}}$ and t other parties in $\mathcal{P}_{\texttt{active}}$.

When generating $[r]_t$, only parties in $\mathcal{T}$ will receive the shares of $[r]_t$. This can be achieved by requiring each dealer only sends shares to parties in $\mathcal{T}$. In the first step of Refresh, parties in $\mathcal{T}$ compute their shares of $[x]_t + [r]_t$ and send them to $P_{\texttt{king}}$. Together with the share held by $P_{\texttt{king}}$, there are $t + 1$ shares, which are enough to reconstruct the whole sharing $[e]_t := [x]_t + [r]_t$. In this way, the cost of generating random sharings for Refresh is reduced by half.

Furthermore, when $P_{\texttt{king}}$ generates $[o]_t$, we can require that the shares of $[o]_t$ held by parties in $\mathcal{P}_{\texttt{active}} \backslash \mathcal{T}$ are set to be 0. Recall that $P_{\texttt{king}}$ learns the shares of $[x]_t$ held by parties in $\mathcal{C}orr$ and the shares of $[o]_t$ held by

parties in $\mathcal{C}orr$ are the same as those of $[x]_t$. Since the shares held by parties in $\mathcal{P}\backslash\mathcal{T}$ are fixed and $|\mathcal{P}\backslash\mathcal{T}| = t$, with these t shares and the secret value $o = 0$, $P_{\texttt{king}}$ can compute the shares of $[o]_t$ held by parties in $\mathcal{T}$. Now, $P_{\texttt{king}}$ only needs to distribute $[o]_t$ to parties in $\mathcal{T}$, and, parties in $\mathcal{P}_{\texttt{active}}\backslash\mathcal{T}$ simply set their shares of $[o]_t$ to be 0. In this way, the cost of distributing $[o]_t$ is reduced by half.

In the DN multiplication protocol, $P_{\texttt{king}}$ can set the shares of $[e]_t$ held by parties in $\mathcal{P}\backslash\mathcal{T}$ to be 0. With these $|\mathcal{P}\backslash\mathcal{T}| = t$ shares and the secret value e, $P_{\texttt{king}}$ can recover the whole sharing $[e]_t$. In this way, $P_{\texttt{king}}$ only needs to distribute $[e]_t$ to parties in $\mathcal{T}$, and, parties in $\mathcal{P}_{\texttt{active}}\backslash\mathcal{T}$ simply set their shares of $[e]_t$ to be 0. As a result, the cost of distributing $[e]_t$ is reduced by half. Note that in the overall protocol, several multiplication gates will be evaluated in parallel, and this optimization can potentially lead to a reduction in the overall communication by a factor of 1/2.

In summary, when $\mathcal{C}orr = \emptyset$, there is no need to run the "Surgery". Our approach achieves 5.5 field elements per multiplication gate, as that in [GS20]. When at least one party is identified as a corrupted party, our approach needs 7.5 field elements per multiplication gate.

Further Problems. We point out that the above approach does not guarantee the correctness. In particular, we need to verify Refresh in the end of the evaluation of each segment. *It is worth noting that the verification of* Refresh *also utilize the virtual transcript idea.* We refer the readers to Section 4 for more details.

Another problem is that we need to make sure adding the surgery procedure in the protocol *will not* break the security of [GS20]. In fact, the security of [GS20] relies on the fact that the DN protocol provides perfect privacy before the output phase even when the adversary is fully malicious. *Replacing the DN protocol by another semi-honest protocol in [GS20] may break down the security entirely.* We refer the readers to Section 6 for more details.

Removing Higher Order Circuit Dependent Terms. We note that the construction from [BSFO12] uses Beaver triples to compute multiplications in the computation phase. One benefit of this method is that Beaver triples provide plenty of redundancy which simplifies the checking process in the computation phase. However, the use of Beaver triples unfortunately requires a verification for each layer of the circuit, which leads to the quadratic term related to D_M.

On the other hand, although when instantiating the transformation from [IKP+16] with the best-known protocol for security with abort, the quadratic term w.r.t. the circuit depth is eliminated, it introduces a new higher order term related to the circuit width. This is because the transformation needs to change the corruption threshold whenever a new corrupted party is identified, which requires an expensive re-sharing process for the input sharings of each segment.

As a summary, we start from the best-known secure-with-abort protocol [GS20], which does not make use of Beaver triples, to remove the quadratic term related to D_M. To avoid the expensive re-sharing process, we rely on a small surgery to proceed. Combining these two ideas, we remove both the higher order terms related to the circuit depth and the circuit width.

2.4 An Omitted Problem: Verifiable System for Checkpoints

To allow all parties to restart the computation from a checkpoint, i.e., the end of the last segment, all the output sharings of the last segment should be verifiable. This is also a problem we omit when checking the virtual transcript: If all parties finally find out that one of the input sharings is inconsistent, then there is no way to identify a new corrupted party or a new pair of disputed parties by only examining the transcript in this segment. This is because the failure comes from the sharings computed in the previous segment.

Therefore, we borrow the idea from [BSFO12] to add verifiability to the output sharings of each segment. At a high-level, for every pair of parties (P_v, P_i) where P_v acts as a verifier, P_v will generate an authentication key (μ, ν) and P_i will receive an authentication tag $\tau = \mu \cdot \mathsf{share}_i + \nu$ of its share share_i. The authentication tag is computed using an MPC protocol. At a later point, P_v can verify the shares of P_i by asking P_i to send the associated authentication tags. Since a wrong share will be rejected by at least $t + 1$ honest parties and

a correct share will be rejected by at most t corrupted parties, a majority vote can decide whether a share is correct or not.

In [BSFO12], each authentication tag is used to authenticate a batch of shares. As a result, the communication cost is independent of the number of shares and therefore, does not affect the concrete efficiency per gate. We make a further improvement to this idea to achieve a larger size of batching by reusing the authentication keys. Some modifications in the verification of authentication tags are also necessary to fit this improvement. We refer the readers to Section 5 for more details.

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. Beyond that, we also assume the existence of a secure broadcast channel, which is available to all parties. The communication complexity is measured by the number of bits X via private channels plus the number of bits Y via the broadcast channel, i.e., $X + Y \cdot \mathcal{BC}$.

We focus on functions which can be represented as arithmetic circuits over a finite field $\mathbb{F}$ (with $|\mathbb{F}| \geq n+1$) with input, addition, multiplication, random, and output gates. Let $\phi = \log |\mathbb{F}|$ be the size of an element in $\mathbb{F}$. We use κ to denote the security parameter and let $\mathbb{K}$ be an extension field of $\mathbb{F}$ (with $|\mathbb{K}| \geq 2^\kappa$). For simplicity, we use κ to denote the size of an element in $\mathbb{K}$. We assume κ to be a multiple of n. Let $e = [\mathbb{K} : \mathbb{F}]$ be the extension degree. We also fix an $\mathbb{F}$-linear bijection $\mathbb{F}^e \to \mathbb{K}$ so that every vector $(s^1, \ldots, s^e) \in \mathbb{F}^e$ maps to an element $\sigma \in \mathbb{K}$.

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 $n = 2t + 1$.

Each party P_i is assigned with 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 gates, multiplication gates, random gates and output gates respectively. We set $C = c_I + c_M + c_R + c_O$ to be the size of the circuit.

3.2 Dispute Control

Dispute control was first introduced in [BTH06]. It is a general strategy to achieve unconditional security efficiently.

The basic idea is to divide the circuit into several segments. For each segment, all parties first evaluate this segment and then check the correctness of the evaluation. After the check is completed, all parties reach a consensus on whether this segment is successfully evaluated.

- If the evaluation is successful, all parties continue to evaluate the next segment.
- Otherwise, the computation for the current segment is discarded and all parties re-evaluate this segment.

To avoid failures due to the same reasons, for each failure of evaluation, all parties run another protocol to locate a pair of two parties, which we refer to as a pair of *disputed parties*, such that at least one of them is corrupted.

To keep track of the identified disputed parties, all parties publicly maintain two sets, $\mathcal{C}orr$ and $\mathcal{D}isp$, which are initially set to be empty. If a new pair of disputed parties is identified, all parties add this pair into $\mathcal{D}isp$. The protocol should guarantee that the same pair of disputed parties will not be identified again. If a party is disputed with at least $(t + 1)$ parties, this party is identified as a corrupted party and then added into $\mathcal{C}orr$. For every party in $\mathcal{C}orr$, since it has been identified as a corrupted party, *every other party is considered to be disputed with this party.* The re-evaluation will use the updated $\mathcal{C}orr$ and $\mathcal{D}isp$.

Therefore, each failure results in an increase in the size of $\mathcal{D}isp$ and only a bounded number ($O(n^2)$) of failures may happen.

In this work, we partition the whole circuit into n^2 segments and the size of each segment is $m = C/n^2$. Let $\mathcal{D}isp_i$ denote the set of parties which are disputed with P_i. Note that $\mathcal{C}orr \subseteq \mathcal{D}isp_i$. We use $\mathcal{P}_{\texttt{active}} = \mathcal{P} \backslash \mathcal{C}orr$ to denote the set of parties which are not identified as corrupted parties currently. Only parties in $\mathcal{P}_{\texttt{active}}$ can participate in the remaining computation. Hereafter, we use *all parties* to refer to parties in $\mathcal{P}_{\texttt{active}}$.

3.3 Secret Sharing

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

A *degree-d* Shamir sharing of $w \in \mathbb{F}$ is a vector $(w_1, \ldots, w_n)$ which satisfies that, there exists a polynomial $f(\cdot) \in \mathbb{F}[X]$ of degree at most d such that $f(0) = w$ 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 $[w]_d$.

Properties of the Shamir's Secret Sharing Scheme. In the following, we will utilize two properties of the Shamir's 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.
$$\forall\ [x]_d, [y]_d,\ [x \cdot y]_{2d} = [x]_d \cdot [y]_d.$$

As [BSFO12], when a dealer P_d distributes a sharing, all shares belong to parties which are disputed with P_d are set to be 0. In this way, two parties that are disputed with each other do not need to communicate when distributing sharings.

When the communication is necessary between disputed parties, a third party which is not disputed with either of parties, referred to as a *relay*, helps pass messages from one party to the other. Note that, for every party $P_i \notin \mathcal{C}orr$, the number of parties which are disputed with P_i is at most t. Therefore, for every pair of disputed parties (P_i, P_j) where $P_i, P_j \notin \mathcal{C}orr$, there is at least one party which can be the relay of (P_i, P_j). We use $P_{i \leftrightarrow j} = P_{j \leftrightarrow i}$ to denote the first party which is not disputed with P_i, P_j, referred to as the relay between P_i and P_j.

3.4 Generating Challenge

We introduce a simple protocol Challenge, which comes from [BSFO12], to let all parties generate an element in $\mathbb{K}$ with high min-entropy. The communication complexity of Challenge is $O(\kappa) \cdot \mathcal{BC}$.

Procedure 1: Challenge

1. Each party $P_i \in \mathcal{P}_{\texttt{active}}$ chooses a random string $\texttt{str}_i \in \{0,1\}^{\frac{\log |\mathbb{K}|}{n}}$ and broadcasts $\texttt{str}_i$.
2. All parties set the string $\texttt{str}_i$ of $P_i \in \mathcal{C}orr$ to be $0^{\frac{\log |\mathbb{K}|}{n}} \in \{0,1\}^{\frac{\log |\mathbb{K}|}{n}}$. Then convert $(\texttt{str}_1, \ldots, \texttt{str}_n)$ into an element λ in $\mathbb{K}$.

Lemma 1 ([BSFO12]). *For any fixed set of t corrupted parties and for any given subset $\mathcal{S} \in \mathbb{K}$, the probability that a challenge generated by* Challenge *lies in $\mathcal{S}$ is at most*
$$\frac{|\mathcal{S}|}{2^{(t+1)\frac{\log |\mathbb{K}|}{n}}} \leq \frac{|\mathcal{S}|}{2^{\kappa/2}}.$$

3.5 Generating Random Sharings

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

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

Protocol 2: RAND

1. Each party $P_i \in \mathcal{P}_{\texttt{active}}$ randomly samples a sharing $[s^{(i)}]_t$ such that the shares held by parties in $\mathcal{D}isp_i$ are set to be 0. Then P_i distributes the shares to other parties. For each $P_i \in \mathcal{C}orr$, all parties take an all-0 sharing as $[s^{(i)}]_t$.
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 $[r^{(1)}]_t, [r^{(2)}]_t, \ldots, [r^{(t+1)}]_t$.

3.6 Generating Random Double Sharings

A pair of double sharings $([r]_t, [r]_{2t})$ is a pair of two sharings of the same secret. One is a degree-t sharing and the other one is a degree-$2t$ sharing. We introduce a simple protocol DOUBLERAND, which comes from [DN07], to let all parties prepare $t+1 = O(n)$ random double sharings in the *semi-honest* setting. The description of DOUBLERAND appears in Protocol 3. The communication complexity of DOUBLERAND is $O(n^2)$ field elements.

Protocol 3: DOUBLERAND

1. Each party $P_i \in \mathcal{P}_{\texttt{active}}$ randomly samples a pair of double sharings $([s^{(i)}]_t, [s^{(i)}]_{2t})$ such that the shares held by parties in $\mathcal{D}isp_i$ are set to be 0. Then P_i distributes the shares to other parties. For each $P_i \in \mathcal{C}orr$, all parties take all-0 sharings as $([s^{(i)}]_t, [s^{(i)}]_{2t})$.
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 note that the randomness of the double sharings is preserved in the *fully malicious* setting. The following lemma is proved in [GS20].

Lemma 2 ([GS20]). *Given the views of* DOUBLERAND *of corrupted parties, all shares of* $[r^{(1)}]_{2t}, \ldots, [r^{(t+1)}]_{2t}$ *held by honest parties are uniformly random.*

4 Evaluation and Verification

In this section, we describe our construction for evaluating one segment of circuit and verifying the computation after evaluation. We start from describing how the protocol proceeds without changing the corruption threshold when identified corrupted parties are kicked out. Then we show how to use virtual transcripts to identify a new corrupted party or a new pair of disputed parties when the computation fails.

4.1 Evaluating a Single Multiplication Gate.

At a high-level, our idea is to compute the shares held by parties in $\mathcal{C}orr$ so that $P_{\texttt{king}}$ can still reconstruct a degree-$2t$ sharing even without receiving shares from parties in $\mathcal{C}orr$. This is done by first doing a small surgery to one of the input sharing $[x]_t$ so that the shares of the resulting sharing $[\tilde{x}]_t$ held by parties in $\mathcal{C}orr$ becomes 0.

In the beginning of the surgery, all parties agree on a set of parties $\mathcal{T}$ such that (1) $|\mathcal{T}| = t+1$, (2) $P_{\texttt{king}} \in \mathcal{T}$, and (3) $\mathcal{T} \bigcap \mathcal{D}isp_{\texttt{king}} = \emptyset$. All parties prepare a random degree-t sharing $[r]_t$ by using Rand with the modification that each dealer only distributes shares to parties in $\mathcal{T}$. Recall that each dealer sets the shares held by parties in $\mathcal{C}orr$ to be 0 in Rand. Since $[r]_t$ is a linear combination of the sharings dealt by each party. The shares of $[r]_t$ held by parties in $\mathcal{C}orr$ are all 0. The description of Refresh appears in Protocol 4.

Protocol 4: Refresh$([x]_t)$

All parties agree on the special party $P_{\texttt{king}}$ and a set of parties $\mathcal{T}$ in the beginning. Let $[r]_t$ be the random sharing which will be used in the protocol. Only parties in $\mathcal{T}$ hold the shares of $[r]_t$.

1. Parties in $\mathcal{T}$ compute $[e]_t := [x]_t + [r]_t$ and send their shares of $[e]_t$ to $P_{\texttt{king}}$.
2. $P_{\texttt{king}}$ reconstructs the whole sharing $[e]_t$ and computes the shares held by parties in $\mathcal{C}orr$.
3. $P_{\texttt{king}}$ generates a degree-t sharing $[o]_t$ such that (1) $o = 0$, (2) the shares held by parties in $\mathcal{P}_{\texttt{active}} \backslash \mathcal{T}$ are set to be 0, and (3) the shares of $[o]_t, [e]_t$ held by parties in $\mathcal{C}orr$ are the same.
4. $P_{\texttt{king}}$ distributes $[o]_t$ to parties in $\mathcal{T}$. Parties in $\mathcal{P}_{\texttt{active}} \backslash \mathcal{T}$ set their shares of $[o]_t$ to be 0.
5. All parties locally compute $[\tilde{x}]_t := [x]_t - [o]_t$.

Then, all parties use the DN multiplication protocol on $([\tilde{x}]_t, [y]_t)$ to compute the result $[x \cdot y]_t$, where $[\tilde{x}]_t$ is the output of Refresh. To this end, all parties first prepare a pair of random double sharings $([r]_t, [r]_{2t})$ by DoubleRand. Recall that each dealer sets the shares held by parties in $\mathcal{C}orr$ to be 0 in DoubleRand. Since $([r]_t, [r]_{2t})$ is a linear combination of the double sharings dealt by each party. The shares of $[r]_t, [r]_{2t}$ held by parties in $\mathcal{C}orr$ are all 0. The description of PartialMult appears in Protocol 5.

Combining Refresh and PartialMult, the description of Mult appears in Protocol 6. The communication complexity of Mult$([x]_t, [y]_t)$ is $O(n)$ elements.

4.2 Evaluating One Segment

In the beginning of each segment, all parties need to prepare enough random sharings for Refresh and random double sharings for PartialMult. All parties first select a special party $P_{\texttt{king}}$ and a set of parties $\mathcal{T}$. Recall that the size of each segment is set to be m.

- For Refresh, we need to prepare m random sharings $[\tilde{r}^{(1)}]_t, \ldots, [\tilde{r}^{(m)}]_t$ such that only parties in $\mathcal{T}$ receive the shares. To this end, all parties invoke $\frac{m}{t+1}$ times of Rand with the modification that each dealer only distributes shares to parties in $\mathcal{T}$.

Protocol 5: PARTIALMULT($[\tilde{x}]_t, [y]_t$)

All parties agree on the special party $P_{\texttt{king}}$ and a set of parties $\mathcal{T}$ in the beginning. Let $([r]_t, [r]_{2t})$ be the random double sharings which will be used in the protocol.

1. All parties locally compute $[e]_{2t} = [\tilde{x}]_t \cdot [y]_t + [r]_{2t}$.
2. $P_{\texttt{king}}$ collects shares of $[e]_{2t}$ from parties in $\mathcal{P}_{\texttt{active}}$ (via relays for parties in $\mathcal{D}isp_{\texttt{king}}$). For each party $P_i \in \mathcal{C}orr$, $P_{\texttt{king}}$ sets the i-th share of $[e]_{2t}$ to be 0. Then $P_{\texttt{king}}$ reconstructs the result e.
3. $P_{\texttt{king}}$ generates a degree-t sharing $[e]_t$ such that the shares held by parties in $\mathcal{P}\backslash\mathcal{T}$ are set to be 0.
4. $P_{\texttt{king}}$ distributes $[e]_t$ to parties in $\mathcal{T}$. Parties in $\mathcal{P}_{\texttt{active}}\backslash\mathcal{T}$ set their shares of $[e]_t$ to be 0.
5. All parties locally compute $[x \cdot y]_t = [e]_t - [r]_t$.

Protocol 6: MULT($[x]_t, [y]_t$)

1. If $\mathcal{C}orr \neq \emptyset$, all parties invoke REFRESH($[x]_t$) and take $[\tilde{x}]_t$ as output. Otherwise, set $[\tilde{x}]_t := [x]_t$.
2. All parties invoke PARTIALMULT($[\tilde{x}]_t, [y]_t$).

– For PARTIALMULT, all parties invoke $\frac{m}{t+1}$ times of DOUBLERAND to prepare $([r^{(1)}]_t, [r^{(1)}]_{2t}), \ldots, ([r^{(m)}]_t, [r^{(m)}]_t)$.

For each addition gate, all parties locally add their shares. For each multiplication gate, all parties invoke MULT. The description of COMPUTE appears in Protocol 7. The communication complexity of COMPUTE(`seg`) is $O(mn\phi + n^2\phi)$ bits.

Protocol 7: COMPUTE(`seg`)

All parties hold the shares of input sharings of `seg`. There are m multiplication gates in `seg`.

1. All parties agree on a special party $P_{\texttt{king}}$ and a set of parties $\mathcal{T}$.
2. If $\mathcal{C}orr \neq \emptyset$, all parties invoke $\frac{m}{t+1}$ times of RAND in $\mathbb{F}$ with the modification that each dealer only distributes shares to parties in $\mathcal{T}$. The output random sharings are denoted by $[\tilde{r}^{(1)}]_t, \ldots, [\tilde{r}^{(m)}]_t$.
3. All parties invoke $\frac{m}{t+1}$ times of DOUBLERAND in $\mathbb{F}$ to prepare $([r^{(1)}]_t, [r^{(1)}]_{2t}), \ldots, ([r^{(m)}]_t, [r^{(m)}]_t)$.
4. Evaluate `seg` as following.
   – For each addition gate with input sharings $[x]_t, [y]_t$, all parties locally compute $[z]_t = [x]_t + [y]_t$.
   – For the k-th multiplication gate with input sharings $[x^{(k)}]_t$ and $[y^{(k)}]_t$, all parties invoke MULT($[x^{(k)}]_t, [y^{(k)}]_t$) with a random sharing $[\tilde{r}^{(k)}]_t$ (if $\mathcal{C}orr \neq \emptyset$) and a pair of random double sharings $([r^{(k)}]_t, [r^{(k)}]_{2t})$.

4.3 Checking the Correctness of Refresh.

We note that the transcript of REFRESH contains 5 degree-t sharings:

$$([x]_t, [\tilde{x}]_t, [\tilde{r}]_t, [e]_t, [o]_t).$$

Here $[x]_t$ is the input sharing, $[\tilde{x}]_t$ is the output sharing, $[\tilde{r}]_t$ is a random sharing which is only held by parties in $\mathcal{T}$, $[e]_t$ is a sharing $P_{\texttt{king}}$ collected from parties in $\mathcal{T}$, and $[o]_t$ is a sharing of 0 dealt by $P_{\texttt{king}}$.

Given m transcripts $\{([x^{(i)}]_t, [\tilde{x}^{(i)}]_t, [\tilde{r}^{(i)}]_t, [e^{(i)}]_t, [o^{(i)}]_t)\}_{i=1}^m$, we want to verify that, for each $i \in [m]$, (1) $x^{(i)} = \tilde{x}^{(i)}$ and (2) the shares of $[\tilde{x}^{(i)}]$ held by parties in $\mathcal{C}orr$ are 0. To this end, we will compress m checks of the transcripts into one check of a single transcript. To protect the privacy of the original m transcripts, we first prepare a random transcript as a mask.

Preparing a Random Transcript of Refresh. All parties first invoke Rand to prepare a random degree-t sharing $[x^{(0)}]_t$, and then prepare a random degree-t sharing $[\tilde{r}^{(0)}]_t$ by Rand with the modification that each dealer only distributes shares to parties in $\mathcal{T}$. Here $[x^{(0)}]_t, [\tilde{r}^{(0)}]_t$ will be used as random masks for $\{[x^{(i)}]_t\}_{i=1}^m$ and $\{[\tilde{r}^{(i)}]_t\}_{i=1}^m$ respectively. Note that we only use one random sharing per call of Rand and the rest of sharings are discarded. Recall that each random sharing generated by Rand is a linear combination of the sharings dealt by each party. At a later point, we may need to reveal the sharings dealt by each party, which will break the privacy of the rest of output sharings by Rand.

The transcript is prepared by invoking Refresh on $[x^{(0)}]_t$ with the random sharing $[\tilde{r}^{(0)}]_t$. Let

$$([x^{(0)}]_t, [\tilde{x}^{(0)}]_t, [\tilde{r}^{(0)}]_t, [e^{(0)}]_t, [o^{(0)}]_t)$$

denote the resulting transcript.

Compressing the Transcripts into One. Consider the following 5 sharings of polynomials:

$$[F(\lambda)]_t = \sum_{i=0}^{m} [x^{(i)}]_t \lambda^i, \quad [\tilde{F}(\lambda)]_t = \sum_{i=0}^{m} [\tilde{x}^{(i)}]_t \lambda^i, \quad [\tilde{R}(\lambda)]_t = \sum_{i=0}^{m} [\tilde{r}^{(i)}]_t \lambda^i,$$
$$[E(\lambda)]_t = \sum_{i=0}^{m} [e^{(i)}]_t \lambda^i, \quad [O(\lambda)]_t = \sum_{i=0}^{m} [o^{(i)}]_t \lambda^i.$$

Note that, by the linear homomorphism property of the Shamir secret sharing scheme, for every λ,

$$([F(\lambda)]_t, [\tilde{F}(\lambda)]_t, [\tilde{R}(\lambda)]_t, [E(\lambda)]_t, [O(\lambda)]_t)$$

is a transcript of Refresh.

If at least one transcript of the original m transcripts is incorrect, then the number of λ such that $([F(\lambda)]_t, [\tilde{F}(\lambda)]_t, [\tilde{R}(\lambda)]_t, [E(\lambda)]_t, [O(\lambda)]_t)$ is a correct transcript is bounded by m. Therefore, to verify the original m transcripts, it is sufficient to examine the transcript $([F(\lambda)]_t, [\tilde{F}(\lambda)]_t, [\tilde{R}(\lambda)]_t, [E(\lambda)]_t, [O(\lambda)]_t)$ for a random $\lambda \in \mathbb{K}$. The description of Check-Refresh appears in Protocol 8. The communication complexity of Check-Refresh is $O(n^2\kappa)$ bits plus $O(n\kappa) \cdot \mathcal{BC}$.

The protocol Check-Refresh needs to invoke two other protocols Analyze-Sharing and Check-Rand to find errors when the transcript is incorrect. Here the protocol Analyze-Sharing is used to handle an inconsistent degree-t sharing. More details can be found in Section 5. The protocol Check-Rand will be introduced later.

Lemma 3. *If all parties broadcast* `ok` *in the end of* Check-Refresh*, with overwhelming probability, for all $i \in \{1, \ldots, m\}$, $x^{(i)} = \tilde{x}^{(i)}$ and the shares of $[\tilde{x}^{(i)}]_t$ held by parties in $\mathcal{C}orr$ are* 0*.*

Handling Incorrect $[\tilde{r}]_t$. Recall that for each $i \in [m]$, $[\tilde{r}^{(i)}]_t$ is a linear combination of the sharings generated by each party. Since $[\tilde{r}]_t$ is a linear combination of $\{[\tilde{r}^{(i)}]_t\}_{i=1}^m$, we can decompose $[\tilde{r}]_t$ into the following form:

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

where $[\tilde{r}(i)]_t$ is a linear combination of the sharings dealt by P_i. To handle an incorrect $[\tilde{r}]_t$, we can examine each $[\tilde{r}(i)]_t$.

The description of Check-Rand appears in Protocol 9. The communication complexity of Check-Rand is $O(n^2\kappa)$ bits plus $O(\kappa) \cdot \mathcal{BC}$.

Protocol 8: CHECK-REFRESH

1. **Preparing A Random Transcript.**
   (a) All parties invoke RAND in $\mathbb{K}$ to prepare a random degree-t sharing $[x^{(0)}]_t$.
   (b) All parties prepare a random degree-t sharing $[\tilde{r}^{(0)}]_t$ by RAND in $\mathbb{K}$ with the modification that each dealer only distributes shares to parties in $\mathcal{T}$.
   (c) All parties invoke REFRESH on $[x^{(0)}]_t$ with random sharing $[\tilde{r}^{(0)}]_t$. Let $([x^{(0)}]_t, [\tilde{x}^{(0)}]_t, [\tilde{r}^{(0)}]_t, [e^{(0)}]_t, [o^{(0)}]_t)$ denote the resulting transcript.
2. All parties invoke CHALLENGE to generate $\lambda \in \mathbb{K}$.
3. All parties compute

$$([x]_t, [\tilde{x}]_t, [\tilde{r}]_t, [e]_t, [o]_t) = \sum_{i=0}^{m} ([x^{(i)}]_t, [\tilde{x}^{(i)}]_t, [\tilde{r}^{(i)}]_t, [e^{(i)}]_t, [o^{(i)}]_t)\lambda^i.$$

4. All parties broadcast their shares of $[x]_t, [\tilde{x}]_t, [o]_t$. Parties in $\mathcal{T}$ broadcast their shares of $[\tilde{r}]_t, [e]_t$. $P_{\texttt{king}}$ broadcasts the sharing $[e]_t$ it received and $[o]_t$ it distributed.
5. All parties check the following.
   - If $[x]_t$ is inconsistent, all parties invoke ANALYZE-SHARING to identify a new corrupted party or a new pair of disputed parties.
   - If the shares of $[\tilde{r}]_t$ held by parties in $\mathcal{C}orr$ are not 0, all parties invoke CHECK-RAND to identify a new corrupted party or a new pair of disputed parties.
   - If some party $P_i \in \mathcal{P}_{\texttt{active}}$ does not follow the protocol, P_i is identified as a new corrupted party.
   - If $P_{\texttt{king}}$ and some party $P_i \in \mathcal{T}$ do not agree on the value sent from one party to the other, $(P_{\texttt{king}}, P_i)$ is identified as a new pair of disputed parties.

   If a new corrupted party or a new pair of disputed party is identified, all parties update $\mathcal{C}orr$ and $\mathcal{D}isp$. If none of above cases happens, all parties take `ok` as output.

Protocol 9: CHECK-RAND

Let $[\tilde{r}]_t = \sum_{i=1}^{n} [\tilde{r}(i)]_t$ be the decomposition of $[\tilde{r}]_t$.

1. Each party $P_i \in \mathcal{P}_{\texttt{active}}$ sends the sharing $[\tilde{r}(i)]_t$ to $P_{\texttt{king}}$ (via a relay if necessary).
2. Each party $P_i \in \mathcal{T}$ sends to $P_{\texttt{king}}$ their shares of $\{[\tilde{r}(j)]_t\}_{j=1}^{n}$ it received.
3. $P_{\texttt{king}}$ checks the following.
   - If $P_{\texttt{king}}$ observes that some party P_i does not follow the protocol, $P_{\texttt{king}}$ broadcasts $(\texttt{accuse}, P_i)$. In the case that $(P_{\texttt{king}}, P_i) \notin \mathcal{D}isp$, $(P_{\texttt{king}}, P_i)$ is a new pair of disputed parties. Otherwise, the relay $P_{\texttt{king}\leftrightarrow i}$ checks the messages it passed from P_i to $P_{\texttt{king}}$ and broadcasts its opinion. If $P_{\texttt{king}\leftrightarrow i}$ agrees, $(P_{\texttt{king}\leftrightarrow i}, P_i)$ is a new pair of disputed parties. Otherwise, $(P_{\texttt{king}\leftrightarrow i}, P_{\texttt{king}})$ is a new pair of disputed parties.
   - If $P_{\texttt{king}}$ observes that $P_i \in \mathcal{P}_{\texttt{active}}, P_j \in \mathcal{T}$ do not agree on the value u sent from one party to the other, $P_{\texttt{king}}$ broadcast $(\texttt{open}, P_i, u, P_j, u')$ where u is the value received from P_i and u' is the value received from P_j. Then P_i, P_j and the relay $P_{\texttt{king}\leftrightarrow i}$ (if used) broadcast the value u. The pair of two parties, where one party directly sends message to the other but two parties do not agree on the same value, is regarded as a new pair of disputed parties.

4.4 Checking the Correctness of PartialMult.

We will use the technique in [GS20] to verify a batch of multiplications. We refer the readers to Section 2 for a sketch of our method.

We first describe the extension of the DN multiplication protocol, which we refer to as Extend-PartialMult, in Protocol 10. Here the input sharing $[\tilde{\boldsymbol{x}}]_t$ satisfies that $\tilde{\boldsymbol{x}} = \boldsymbol{x}$ and the shares held by parties in $\mathcal{C}orr$ are 0.

Protocol 10: Extend-PartialMult$([\tilde{\boldsymbol{x}}]_t, [\boldsymbol{y}]_t)$

All parties agree on the special party $P_{\texttt{king}}$ and a set of parties $\mathcal{T}$ in the beginning. Let $([r]_t, [r]_{2t})$ be the random double sharings which will be used in the protocol.

1. All parties locally compute $[e]_{2t} = [\tilde{\boldsymbol{x}}]_t \odot [\boldsymbol{y}]_t + [r]_{2t}$.
2. $P_{\texttt{king}}$ collects shares of $[e]_{2t}$ from parties in $\mathcal{P}_{\texttt{active}}$ (via relays for parties in $\mathcal{D}isp_{\texttt{king}}$). For each party $P_i \in \mathcal{C}orr$, $P_{\texttt{king}}$ sets the i-th share of $[e]_{2t}$ to be 0. Then $P_{\texttt{king}}$ reconstructs the result e.
3. $P_{\texttt{king}}$ generates a degree-t sharing $[e]_t$ such that the shares held by parties in $\mathcal{P}\backslash\mathcal{T}$ are set to be 0.
4. $P_{\texttt{king}}$ distributes $[e]_t$ to parties in $\mathcal{T}$. Parties in $\mathcal{P}_{\texttt{active}}\backslash\mathcal{T}$ set their shares of $[e]_t$ to be 0.
5. All parties locally compute $[\boldsymbol{x} \odot \boldsymbol{y}]_t = [e]_t - [r]_t$.

The transcript of Extend-PartialMult is denoted by

$$([\tilde{\boldsymbol{x}}]_t, [\boldsymbol{y}]_t, [r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t).$$

We now describe the extension of the Batch-wise Multiplication Verification technique in [BSFO12] which can compress the check of m transcripts of inner-product tuples into a check of a single transcript. The description of Compress appears in Protocol 11. The communication complexity of Compress is $O(mn\kappa)$ bits plus $O(\kappa) \cdot \mathcal{BC}$.

The verification of the original m multiplication tuples in [GS20] is consist of the following three steps.

Step One: De-Linearization. The first step is transforming a batch of m multiplication tuples into an inner-product tuple of dimension m. The description of De-Linearization appears in Protocol 12. The communication complexity of De-Linearization is $O(\kappa) \cdot \mathcal{BC}$.

Step Two: Dimension-Reduction. The second step is to reduce the dimension of the inner-product tuple from De-Linearization. The description of Dimension-Reduction appears in Protocol 13. The communication complexity of Dimension-Reduction is $O(n\kappa^2)$ bits plus $O(\kappa) \cdot \mathcal{BC}$.

Step Three: Randomization. In the final step, all parties prepare a random transcript of PartialMult and add this transcript in the last call of Compress. The description of Randomization appears in Protocol 14. The communication complexity of Randomization is $O(n^2\kappa + n\kappa^2)$ bits plus $O(\kappa) \cdot \mathcal{BC}$.

Checking the Final Transcript. The transcript of the final multiplication protocol is denoted by

$$([\alpha]_t, [\beta]_t, [\delta]_t, [\delta]_{2t}, [\eta]_{2t}, [\eta]_t, [\gamma]_t).$$

The description of Check-Single-Mult appears in Protocol 15. The communication complexity of Check-Single-Mult is $O(n\kappa) \cdot \mathcal{BC}$ (not counting Analyze-Sharing).

The protocol Check-Single-Mult needs to invoke two other protocols Analyze-Sharing and Check-DoubleRand to find errors when the transcript is incorrect. Here the protocol Analyze-Sharing is used to handle an inconsistent degree-t sharing. More details can be found in Section 5. The protocol Check-DoubleRand will be introduced later.

Protocol 11: COMPRESS

Let $\{([\tilde{\boldsymbol{x}}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^m$ denote the transcripts of the m inner-product tuples.

1. Let $[\tilde{\boldsymbol{F}}(\cdot)]_t, [\boldsymbol{G}(\cdot)]_t$ be two sharings of vectors of polynomials of degree $(m-1)$ such that
$$\forall i \in [m]: \ [\tilde{\boldsymbol{F}}(i)]_t = [\tilde{\boldsymbol{x}}^{(i)}]_t, \ [\boldsymbol{G}(i)]_t = [\boldsymbol{y}^{(i)}]_t.$$
All parties can use $\{([\tilde{\boldsymbol{x}}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t)\}_{i=1}^m$ to compute $[\tilde{\boldsymbol{F}}(\cdot)]_t, [\boldsymbol{G}(\cdot)]_t$ locally.
2. All parties compute $[\tilde{\boldsymbol{x}}^{(i)}]_t = [\tilde{\boldsymbol{F}}(i)]_t, [\boldsymbol{y}^{(i)}]_t = [\boldsymbol{G}(i)]_t$ for all $i \in \{m+1, \ldots, 2m-1\}$.
3. For all $i \in \{m+1, \ldots, 2m-1\}$, all parties invoke EXTEND-PARTIALMULT on $[\tilde{\boldsymbol{x}}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t$. Let
$$([\tilde{\boldsymbol{x}}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)$$
denote the transcript.
4. Let $[R(\cdot)]_t, [R(\cdot)]_{2t}, [E(\cdot)]_{2t}, [E(\cdot)]_t, [H(\cdot)]_t$ be sharings of polynomials of degree $2(m-1)$ such that
$$\begin{aligned}\forall i \in [2m-1]: \ & [R(i)]_t = [r^{(i)}]_t, \ [R(i)]_{2t} = [r^{(i)}]_{2t}, \ [E(i)]_{2t} = [e^{(i)}]_{2t}, \\ & [E(i)]_t = [e^{(i)}]_t, \ [H(i)]_t = [z^{(i)}]_t.\end{aligned}$$
All parties can use $\{([r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t)\}_{i=1}^{2m-1}$ to compute these sharings locally.
5. All parties invoke CHALLENGE to generate $\lambda \in \mathbb{K}$.
6. All parties output
$$([\tilde{\boldsymbol{F}}(\lambda)]_t, [\boldsymbol{G}(\lambda)]_t, [R(\lambda)]_t, [R(\lambda)]_{2t}, [E(\lambda)]_{2t}, [E(\lambda)]_t, [H(\lambda)]_t).$$

Protocol 12: DE-LINEARIZATION

Let $\{([\tilde{x}^{(i)}]_t, [y^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^m$ denote the transcripts of the m multiplication tuples.

1. All parties invoke CHALLENGE to generate $\lambda \in \mathbb{K}$.
2. All parties set
$$\begin{aligned}[\tilde{\boldsymbol{x}}]_t &= ([\tilde{x}^{(1)}]_t, \lambda[\tilde{x}^{(2)}]_t, \ldots, \lambda^{m-1}[\tilde{x}^{(m)}]_t) \\ [\boldsymbol{y}]_t &= ([y^{(1)}]_t, [y^{(2)}]_t, \ldots, [y^{(m)}]_t) \\ ([r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t) &= \sum_{i=1}^m \lambda^{i-1}([r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t),\end{aligned}$$
and output $([\tilde{\boldsymbol{x}}]_t, [\boldsymbol{y}]_t, [r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t)$.

Handling Incorrect $([\delta]_t, [\delta]_{2t})$. Note that $([\delta]_t, [\delta]_{2t})$ can be decomposed into the following form:

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

where $([\delta(i)]_t, [\delta(i)]_{2t})$ is a linear combination of the double sharings dealt by P_i. To handle an incorrect pair of double sharings $([\delta]_t, [\delta]_{2t})$, we will examine each $([\delta(i)]_t, [\delta(i)]_{2t})$. The description of CHECK-DOUBLERAND appears in Protocol 16. The communication complexity of CHECK-DOUBLERAND is $O(n^2\kappa)$ bits plus $O(\kappa) \cdot \mathcal{BC}$.

Protocol 13: DIMENSION-REDUCTION

1. Let $([\tilde{\boldsymbol{x}}]_t, [\boldsymbol{y}]_t, [r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t)$ denote the transcript of the inner-product tuple of dimension m.
2. All parties partition $[\tilde{\boldsymbol{x}}]_t, [\boldsymbol{y}]_t$ into κ equal pieces as following:
$$[\tilde{\boldsymbol{x}}]_t = ([\tilde{\boldsymbol{x}}^{(1)}]_t, [\tilde{\boldsymbol{x}}^{(2)}]_t, \ldots, [\tilde{\boldsymbol{x}}^{(\kappa)}]_t), \quad [\boldsymbol{y}]_t = ([\boldsymbol{y}^{(1)}]_t, [\boldsymbol{y}^{(2)}]_t, \ldots, [\boldsymbol{y}^{(\kappa)}]_t).$$
3. For every $i \in [\kappa - 1]$, all parties invoke EXTEND-PARTIALMULT with input $[\tilde{\boldsymbol{x}}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t$ to compute $[z^{(i)}]_t$. The transcript is denoted by $([\tilde{\boldsymbol{x}}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)$.
4. All parties compute
$$\begin{aligned}&([r^{(\kappa)}]_t, [r^{(\kappa)}]_{2t}, [e^{(\kappa)}]_{2t}, [e^{(\kappa)}]_t, [z^{(\kappa)}]_t)\\ &= ([r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t) - \sum_{i=1}^{\kappa-1} ([r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)\end{aligned}$$
and regard $([\tilde{\boldsymbol{x}}^{(\kappa)}]_t, [\boldsymbol{y}^{(\kappa)}]_t, [r^{(\kappa)}]_t, [r^{(\kappa)}]_{2t}, [e^{(\kappa)}]_{2t}, [e^{(\kappa)}]_t, [z^{(\kappa)}]_t)$ as the transcript of EXTEND-PARTIALMULT.
5. Invoke COMPRESS on $\{([\tilde{\boldsymbol{x}}^{(i)}]_t, [\boldsymbol{y}^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^{\kappa}$. The resulting transcript of the inner-product tuple is of dimension m/κ.

Protocol 14: RANDOMIZATION

1. **Preparing A Random Transcript.**
   (a) All parties invoke RAND in $\mathbb{K}$ to prepare random degree-t sharings $[\tilde{x}^{(0)}]_t, [y^{(0)}]_t$. All parties invoke DOUBLERAND in $\mathbb{K}$ to prepare a pair of random double sharings $([r^{(0)}]_t, [r^{(0)}]_{2t})$. Only one pair of random double sharings is used in DOUBLERAND and the rest of sharings are discarded.
   (b) All parties invoke PARTIALMULT on $[\tilde{x}^{(0)}]_t, [y^{(0)}]_t$ with random double sharings $([r^{(0)}]_t, [r^{(0)}]_{2t})$. The resulting transcript is denoted by $([\tilde{x}^{(0)}]_t, [y^{(0)}]_t, [r^{(0)}]_t, [r^{(0)}]_{2t}, [e^{(0)}]_{2t}, [e^{(0)}]_t, [z^{(0)}]_t)$.
2. Let $([\tilde{\boldsymbol{x}}]_t, [\boldsymbol{y}]_t, [r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t)$ denote the transcript of the inner-product tuple of dimension κ.
3. All parties interpret $[\tilde{\boldsymbol{x}}]_t, [\boldsymbol{y}]_t$ as
$$[\tilde{\boldsymbol{x}}]_t = ([\tilde{x}^{(1)}]_t, [\tilde{x}^{(2)}]_t, \ldots, [\tilde{x}^{(\kappa)}]_t), \quad [\boldsymbol{y}]_t = ([y^{(1)}]_t, [y^{(2)}]_t, \ldots, [y^{(\kappa)}]_t).$$
4. For each $i \in [\kappa - 1]$, all parties invoke PARTIALMULT on $[\tilde{x}^{(i)}]_t, [y^{(i)}]_t$ to compute $[z^{(i)}]_t$ where $z^{(i)} = x^{(i)} \cdot y^{(i)}$. The transcript is denoted by $([\tilde{x}^{(i)}]_t, [y^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)$.
5. All parties compute
$$\begin{aligned}&([r^{(\kappa)}]_t, [r^{(\kappa)}]_{2t}, [e^{(\kappa)}]_{2t}, [e^{(\kappa)}]_t, [z^{(\kappa)}]_t)\\ &= ([r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t) - \sum_{i=1}^{\kappa-1} ([r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)\end{aligned}$$
and regard $([\tilde{x}^{(\kappa)}]_t, [y^{(\kappa)}]_t, [r^{(\kappa)}]_t, [r^{(\kappa)}]_{2t}, [e^{(\kappa)}]_{2t}, [e^{(\kappa)}]_t, [z^{(\kappa)}]_t)$ as the transcript of PARTIALMULT.
6. Invoke COMPRESS on $\{([\tilde{x}^{(i)}]_t, [y^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^{\kappa}$. The output is denoted by $([\alpha]_t, [\beta]_t, [\delta]_t, [\delta]_{2t}, [\eta]_{2t}, [\eta]_t, [\gamma]_t)$.

4.5 Summary of the Evaluation

Now we are ready to present the whole protocol for evaluating each segment, which is shown in Protocol 17. The communication complexity of EVAL(seg) is $O(mn\phi + n^2\kappa + n\kappa^2)$ bits plus $O(n\kappa) \cdot \mathcal{BC}$.

Protocol 15: CHECK-SINGLE-MULT

1. Each party broadcasts its shares of $[\alpha]_t, [\beta]_t, [\delta]_t, [\delta]_{2t}, [\eta]_{2t}, [\eta]_t, [\gamma]_t$. $P_{\texttt{king}}$ broadcasts the sharing $[\eta]_{2t}$ it received and $[\eta]_t$ it distributed.
2. All parties check the following.
   - If either of $[\alpha]_t, [\beta]_t$ is inconsistent, all parties invoke ANALYZE-SHARING to identify a new corrupted party or a new pair of disputed parties.
   - If $([\delta]_t, [\delta]_{2t})$ is not a valid pair of double sharings, all parties invoke CHECK-DOUBLERAND to identify a new corrupted party or a new pair of disputed parties.
   - If some party $P_i \in \mathcal{P}_{\texttt{active}}$ does not follow the protocol, P_i is identified as a new corrupted party.
   - If $P_{\texttt{king}}$ and some party $P_i \in \mathcal{P}_{\texttt{active}}$ do not agree on the value sent from one party to the other, there are two cases:
     - If $(P_{\texttt{king}}, P_i) \notin \mathcal{D}isp$, $(P_{\texttt{king}}, P_i)$ is identified as a new pair of disputed parties.
     - Otherwise, the relay $P_{\texttt{king}\leftrightarrow i}$ publishes the value it helped pass. If P_i and $P_{\texttt{king}\leftrightarrow i}$ publishes different values, $(P_{\texttt{king}\leftrightarrow i}, P_i)$ is identified as a new pair of disputed parties. Otherwise, $(P_{\texttt{king}\leftrightarrow i}, P_{\texttt{king}})$ is identified as a new pair of disputed parties.

   If a new corrupted party or a new pair of disputed party is identified, all parties update $\mathcal{C}orr$ and $\mathcal{D}isp$. If none of above cases happens, all parties take `ok` as output.

Protocol 16: CHECK-DOUBLERAND

Let $([\delta]_t, [\delta]_{2t}) = \sum_{i=1}^{n} ([\delta(i)]_t, [\delta(i)]_{2t})$ be the decomposition of $([\delta]_t, [\delta]_{2t})$.

1. Each party $P_i \in \mathcal{P}_{\texttt{active}}$ sends the double sharings $([\delta(i)]_t, [\delta(i)]_{2t})$ to $P_{\texttt{king}}$ (via a relay if necessary).
2. Each party $P_i \in \mathcal{P}_{\texttt{active}}$ sends to $P_{\texttt{king}}$ their shares of $\{([\delta(j)]_t, [\delta(j)]_{2t})\}_{j=1}^{n}$ it received.
3. $P_{\texttt{king}}$ checks the following.
   - If $P_{\texttt{king}}$ observes that some party P_i does not follow the protocol, $P_{\texttt{king}}$ broadcasts $(\texttt{accuse}, P_i)$. In the case that $(P_{\texttt{king}}, P_i) \notin \mathcal{D}isp$, $(P_{\texttt{king}}, P_i)$ is a new pair of disputed parties. Otherwise, the relay $P_{\texttt{king}\leftrightarrow i}$ checks the messages it passed from P_i to $P_{\texttt{king}}$ and broadcasts its opinion. If $P_{\texttt{king}\leftrightarrow i}$ agrees, $(P_{\texttt{king}\leftrightarrow i}, P_i)$ is a new pair of disputed parties. Otherwise, $(P_{\texttt{king}\leftrightarrow i}, P_{\texttt{king}})$ is a new pair of disputed parties.
   - If $P_{\texttt{king}}$ observes that $P_i, P_j \in \mathcal{P}_{\texttt{active}}$ do not agree on the value u sent from one party to the other, $P_{\texttt{king}}$ broadcast $(\texttt{open}, P_i, u, P_j, u')$ where u is the value received from P_i and u' is the value received from P_j. Then P_i, P_j and the relays $P_{\texttt{king}\leftrightarrow i}, P_{\texttt{king}\leftrightarrow j}$ (if used) broadcast the value u. The pair of two parties, where one party directly sends message to the other but two parties do not agree on the same value, is regarded as a new pair of disputed parties.

5 Verifiable System

As discussed in Section 2, after evaluating a segment successfully, we need to add verifiability to the output sharings of this segment. We observe that for each wire, the associated sharing $[x]_t$ can be decomposed into the following form:

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

where $[x(i)]_t$ is a linear combination of the sharings dealt by P_i. To see this, it is sufficient to show that this property is preserved under evaluation of multiplication gates and addition gates.

First note that this property is preserved under linear combination, i.e. if $[x]_t, [y]_t$ can be decomposed into the above form, then any linear combination of $[x]_t, [y]_t$ can also do. Therefore, this property is preserved when evaluating addition gates. As for a multiplication gate, note that the output of PARTIALMULT is

Protocol 17: EVAL(seg)

1. All parties invoke COMPUTE(seg).
2. All parties invoke CHECK-REFRESH.
   - If a new corrupted party or a new pair of disputed parties is identified, all parties update $\mathcal{C}orr$ and $\mathcal{D}isp$. Then halt.
   - Otherwise, all parties proceed to verify PARTIALMULT.
3. All parties do the following steps to verify PARTIALMULT.
   (a) All parties invoke DE-LINEARIZATION.
   (b) All parties repeatedly invoke DIMENSION-REDUCTION until the dimension of the output inner-product tuple is reduced to κ.
   (c) All parties invoke RANDOMIZATION.
   (d) All parties invoke CHECK-SINGLE-MULT.
      - If a new corrupted party or a new pair of disputed parties is identified, all parties update $\mathcal{C}orr$ and $\mathcal{D}isp$. Then halt.
      - Otherwise, all parties accept the computation of this segment.

$[z]_t = [e]_t - [r]_t$, where $[e]_t$ is a sharing dealt by $P_{\texttt{king}}$ and $[r]_t$ is a sharing generated by DOUBLERAND. Recall that $[r]_t$ generated by DOUBLERAND is a linear combination of sharings dealt by each party. Therefore, $[z]_t$ can also be decomposed into the above form. The property is preserved when evaluating multiplication gates.

With this observation, instead of directly adding verifiability to the output sharings, we can add verifiability to the sharings dealt by each party in this segment. In this way, the dealer is able to help us identify a new corrupted party or a new pair of disputed parties when its sharing is inconsistent. We borrow the idea in [BSFO12] to add verifiability to the sharings dealt by each party. We also improve the communication complexity of this idea, which will be introduced in the following.

5.1 Adding Verifiability to Sharings dealt by Each Party

Before we add verifiability to the sharings dealt by each party, we first check the consistency of these sharings. Although a consistency check is not necessary when evaluate the protocol, it is necessary for the verifiability of sharings since each party will be responsible for the shares it received.

Suppose $[s^{(1)}]_t, [s^{(2)}]_t, \ldots, [s^{(k)}]_t$ are the degree-t sharings dealt by each party in this segment. In our construction, k is of size $O(m)$. The description of VERIFY-SHARING appears in Protocol 18. The communication complexity of VERIFY-SHARING is $O(n^2\kappa)$ bits plus $O(n\kappa) \cdot \mathcal{BC}$ (not counting ANAYLZE-SHARING). The protocol ANALYZE-SHARING is used to handle an inconsistent degree-t sharing, which is introduced in Section 5.

For each party $P_d \in \mathcal{P}_{\texttt{active}}$, let $[s^{(1)}(d)]_t, [s^{(2)}(d)]_t, \ldots, [s^{(\ell)}(d)]_t$ denote the sharings we want to add verifiability. Here ℓ is the size of each batch. In the final protocol, all the sharings dealt by P_d will be handled in a batch way. Since in each segment, P_d dealt $\frac{m}{t+1}$ sharings in COMPUTE(seg), the batch size ℓ will be set to be $O(\frac{m}{t+1})$.

Recall that $\mathbb{K}$ is an extension field of $\mathbb{F}$ and $e = [\mathbb{K} : \mathbb{F}]$ is the extension degree. For each vector of e sharings $([s^{(1)}(d)]_t), \ldots, [s^{(e)}(d)]_t$ in $\mathbb{F}$, it maps to a sharing $[\sigma]_t$ in $\mathbb{K}$. Let $q = \ell/e$. Then the sharings dealt by P_d are transformed into $[\sigma^{(1)}(d)]_t, \ldots, [\sigma^{(q)}(d)]_t$ in $\mathbb{K}$. Let $[\boldsymbol{\sigma}(d)]_t$ denote the vector of sharings $([\sigma^{(1)}(d)]_t, \ldots, [\sigma^{(q)}(d)]_t)$.

After P_d distributes the sharings to other parties, for every pair of parties (P_v, P_i), P_i needs to authenticate its shares of $[\boldsymbol{\sigma}(d)]_t$ to P_v. To this end, P_v prepares a pair of authentication keys $(\boldsymbol{\mu}_{v\to i}, \nu)$ where $\boldsymbol{\mu}_{v\to i} \in \mathbb{K}^q$ and $\nu \in \mathbb{K}$ are chosen randomly. Let $\boldsymbol{\sigma}_i(d)$ denote the shares held by P_i. All parties will engage an MPC

Protocol 18: VERIFY-SHARING

1. All parties prepare a random degree-t sharing $[s^{(0)}]_t$ by RAND in $\mathbb{K}$.
2. All parties invoke CHALLENGE to generate $\lambda \in \mathbb{K}$.
3. All parties compute the sharing:
$$[\sigma]_t = \sum_{i=0}^{k} \lambda^i [s^{(i)}]_t.$$
4. All parties broadcast their shares of $[\sigma]_t$ and check whether $[\sigma]_t$ is consistent.
   - If $[\sigma]_t$ is inconsistent, all parties invoke ANALYZE-SHARING to identify a new corrupted party or a new pair of disputed parties.
   - Otherwise, all parties accept the sharings dealt by each party in this segment.

protocol to compute an authentication tag

$$\tau = \boldsymbol{\mu}_{v\to i} \odot \boldsymbol{\sigma}_i(d) + \nu,$$

and P_i is the only receiver of the tag τ. Note that, without learning the authentication keys $(\boldsymbol{\mu}_{v\to i}, \nu)$, the probability that P_i generates a valid authentication tag for a batch of wrong shares is negligible.

The functionality $\mathcal{F}_{\text{Tag}}$ of this process is described in Functionality 19.

Functionality 19: $\mathcal{F}_{\text{Tag}}(P_d)$

1. On receiving $(\texttt{disputed}, P_i, P_j)$, where $(P_i, P_j) \notin \mathcal{D}isp$ and at least one of P_i, P_j is corrupted, $\mathcal{F}_{\text{Tag}}$ sends $(\texttt{disputed}, P_i, P_j)$ to all parties and halts. On receiving $(\texttt{corrupted}, P_i)$, where $P_i \notin \mathcal{C}orr$ and P_i is corrupted, $\mathcal{F}_{\text{Tag}}$ sends $(\texttt{corrupted}, P_i)$ to all parties and halts.
2. $\mathcal{F}_{\text{Tag}}$ receives the degree-t sharings $\{[s^{(j)}(d)]_t\}_{j=1}^{\ell}$ from P_d. $\mathcal{F}_{\text{Tag}}$ interprets it as $[\boldsymbol{\sigma}(d)]_t$.
3. For each pair of parties (P_v, P_i) such that $(P_v, P_i), (P_i, P_d) \notin \mathcal{D}isp$, $\mathcal{F}_{\text{Tag}}$ does the following.
   (a) If P_v is corrupted, $\mathcal{F}_{\text{Tag}}$ receives $(\boldsymbol{\mu}_{v\to i}, \nu)$ from P_v. Otherwise, $\mathcal{F}_{\text{Tag}}$ randomly samples $(\boldsymbol{\mu}_{v\to i}, \nu)$ for P_v and sends $(\boldsymbol{\mu}_{v\to i}, \nu)$ to P_v.
   (b) $\mathcal{F}_{\text{Tag}}$ computes
   $$\tau = \boldsymbol{\mu}_{v\to i} \odot \boldsymbol{\sigma}_i(d) + \nu$$
   and sends τ to P_i.
   (c) If P_v, P_i are both honest, $\mathcal{F}_{\text{Tag}}$ sends $\boldsymbol{\mu}_{v\to i}$ to the adversary.

Remark 1. In $\mathcal{F}_{\text{Tag}}$, for a pair of honest parties (P_v, P_i), the functionality will reveal the authentication key $\boldsymbol{\mu}_{v\to i}$ to the adversary. This leakage is acceptable since the tags between (P_v, P_i) will never be examined. For the adversary, they are just several random elements in $\mathbb{K}$. However, allowing this leakage simplifies the realization of $\mathcal{F}_{\text{Tag}}$.

5.2 Realization of $\mathcal{F}_{\text{Tag}}$

In this part, we introduce the protocol which realizes $\mathcal{F}_{\text{Tag}}$ presented above.

Recall that $\ell = O(\frac{m}{t+1})$ is the batch size. For a dealer P_d, let $[s^{(1)}(d)]_t$, $[s^{(2)}(d)]_t, \ldots, [s^{(\ell)}(d)]_t$ denote the sharings we want to add verifiability. Let $q = \ell/e$, where $e = [\mathbb{K} : \mathbb{F}]$ is the extension degree. Then these

sharings map to a vector of sharings $[\boldsymbol{\sigma}(d)]_t$ with dimension q in $\mathbb{K}$. Let $\boldsymbol{\sigma}_i(d)$ denote the shares of P_i. For a pair of parties (P_v, P_i), P_v will generate a pair of authentication keys $(\boldsymbol{\mu}_{v\to i}, \nu)$ and our goal is to compute the authentication tag

$$\tau = \boldsymbol{\mu}_{v\to i} \odot \boldsymbol{\sigma}_i(d) + \nu$$

for P_i.

To this end, the high-level idea is to let P_v share its authentication keys to all parties such that *the secrets are hidden at position α_i instead of* 0. We use $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i, \lceil \nu \rfloor_t^i$ to denote these two sharings. Then locally computing $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i \odot [\boldsymbol{\sigma}(d)]_t + \lceil \nu \rfloor_t^i$ yields a degree-$2t$ sharing where the secret hidden at position α_i is τ. To learn the secret, P_i will collects all the shares and reconstruct the secret.

As for a pair of authentication keys $(\boldsymbol{\mu}_{v\to i}, \nu)$, $\boldsymbol{\mu}_{v\to i}$ is served as a long term key, i.e., it is used in many different batches of shares held by P_i. On the other hand, a new ν will be generated per batch. Therefore, the authentication tags are computed in two steps: (1) preparing the sharings of long term keys and (2) computing tags using the above method.

In [BSFO12], the long term keys are generated each segment. However, we note that if all parties behaved honestly in the last segment, then there is no need to change the long term keys. Therefore, our idea is to reuse the long term keys in different segments and only generate a new one if necessary. In this way, we can use a longer key and achieve a larger size of batching. Some modifications in verifying the authentication tags are also necessary to ensure the communication complexity does not blow up.

In the following, we will show how authentication keys are shared in Section 5.3 and then show how to compute the tags in Section 5.4.

5.3 Key Distribution and Maintenance

We first introduce the notion of twisted sharings, which is a variant of Shamir secret sharing scheme.

A *twisted degree-d* sharing of $w \in \mathbb{F}$ with respect to P_j is a vector $(w_1, \ldots, w_{j-1}, w_{j+1}, \ldots, w_n)$ which satisfies that, there exists a polynomial $f(\cdot) \in \mathbb{F}[X]$ of degree at most d such that $f(0) = 0$, $f(\alpha_j) = w$ and $f(\alpha_i) = w_i$ for $i \in \{1, \ldots, n\} \backslash \{j\}$. Each party $P_i \neq P_j$ holds a share w_i and the whole sharing is denoted as $\lceil w \rfloor_d^j$.

Key-Distribution. The first step is let P_v distribute twisted degree-t sharings $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$ for every P_i such that $(P_v, P_i) \notin \mathcal{D}isp$. Recall that $\ell = O(\frac{m}{t+1})$ and the size of $\boldsymbol{\mu}_{v\to i}$ is $q = \ell/e$. The description of KEY-DISTRIBUTION(P_v, P_i) appears in Protocol 20. The communication complexity of KEY-DISTRIBUTION(P_v, P_i) is $O(\ell n \phi)$ bits.

Protocol 20: KEY-DISTRIBUTION(P_v, P_i)

1. P_v randomly samples $\boldsymbol{\mu}_{v\to i} \in \mathbb{K}^q$.
2. P_v randomly samples $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$ such that all shares belong to parties in $\mathcal{D}isp_v$ are set to be 0.
3. For every party $P_j \notin \mathcal{D}isp_v \bigcup \{P_v, P_i\}$, P_v sends the j-th shares of $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$ to P_j.

Checking the Correctness. In this step, we check the validness of all twisted degree-t sharings of the authentication keys. To be more clear, for every twisted degree-t sharing, we will check whether all shares held by honest parties lie on a polynomial $f(\cdot)$ of degree at most t and $f(0) = 0$. The verification is

Protocol 21: CHECK-KEY

For every pair of parties $(P_v, P_i) \notin \mathcal{D}isp$, the following steps are done in parallel:

1. P_v randomly generates $\lceil \mu^{(0)}_{v\to i} \rfloor^i_t$ in $\mathbb{K}$ such that shares belong to parties in $\mathcal{D}isp_v$ are set to be 0.
2. For every party $P_j \notin \mathcal{D}isp_v \bigcup \{P_v, P_i\}$, P_v sends the j-th share of $\lceil \mu^{(0)}_{v\to i} \rfloor^i_t$ to P_j.
3. All parties invoke CHALLENGE and generate $\lambda \in \mathbb{K}$. This step is done only once across executions for all pairs of parties.
4. Let $\lceil \boldsymbol{\mu}_{v\to i} \rfloor^i_t = (\lceil \mu^{(1)}_{v\to i} \rfloor^i_t, \ldots, \lceil \mu^{(q)}_{v\to i} \rfloor^i_t)$. All parties locally compute $\lceil \sigma_{v\to i} \rfloor^i_t = \lceil \mu^{(0)}_{v\to i} \rfloor^i_t + \sum_{k=1}^{q} \lambda^k \lceil \mu^{(k)}_{v\to i} \rfloor^i_t$.
5. For every P_s acting as a verifier, every party $P_j \notin \mathcal{D}isp_s$ sends its share of $\lceil \sigma_{v\to i} \rfloor^i_t$ to P_s. P_s checks whether all shares lie on a polynomial $f(\cdot)$ of degree at most t such that $f(0) = 0$ and the shares held by parties in $\mathcal{D}isp_v$ are 0.
6. For every P_s acting as a verifier, if all checks pass, it broadcasts `ok`. Otherwise, it broadcasts $(\texttt{fault}, v, i)$ to indicate that the check for (P_v, P_i) fails. If multiple checks fail, P_s only broadcasts one of them. This step is done only once after P_s receives all shares in all executions.

done in a similar way to VERIFY-SHARING. The description of CHECK-KEY appears in Protocol 21. The communication complexity of CHECK-KEY is $O(n^4\kappa)$ bits plus $O(n\phi + \kappa) \cdot \mathcal{BC}$.

Combining with Lemma 1, we have the following lemma.

Lemma 4. *If all parties broadcast* `ok` *in the end of* CHECK-KEY*, then with overwhelming probability, all twisted degree-t sharings are valid.*

In the case that some party P_s broadcast $(\texttt{fault}, v, i)$ in the end of CHECK-KEY we need to find a new pair of disputed parties. If multiple parties broadcast `fault`, let P_s be the party with the smallest index.

We first select a reviewer P_r which is not disputed with either P_s or P_v. Since all twisted degree-t sharings are generated by P_v, P_v is able to provide the valid version of $\lceil \sigma_{v\to i} \rfloor^i_t$ to P_r. P_s is required to provide the invalid version of $\lceil \sigma_{v\to i} \rfloor^i_t$ to P_r. Then P_r is able to find the different shares and identify a new pair of disputed parties. The description of FL-KEY appears in Protocol 22. The communication complexity of FL-KEY is $O(n\kappa)$ bits plus $O(\kappa) \cdot \mathcal{BC}$.

Protocol 22: FL-KEY

1. If $(P_v, P_s) \in \mathcal{D}isp$, let $P_r := P_{v\leftrightarrow s}$. Otherwise, let $P_r := P_v$.
2. P_s and P_v send the twisted degree-t sharing $\lceil \sigma_{v\to i} \rfloor^i_t$ to P_r.
3. P_r checks the following:
   - If the sharing provided by P_v is *invalid*, P_r broadcasts $(\texttt{accuse}, v)$. All parties regard (P_v, P_r) as a new pair of disputed parties.
   - If the sharing provided by P_s is *valid*, P_r broadcasts $(\texttt{accuse}, s)$. All parties regard (P_s, P_r) as a new pair of disputed parties.
   - If neither of above cases happens, P_r finds the party P_j where the j-th shares of these two twisted degree-t sharings are different. Let σ_j, σ'_j denote the j-th shares sent by P_v and P_s respectively.
     (a) P_r broadcasts $(\texttt{incorrect}, j, \sigma_j, \sigma'_j)$.
     (b) P_v, P_s, P_j broadcast the j-th shares of $\lceil \sigma_{v\to i} \rfloor^i_t$ they sent/received.
     (c) The pair of parties, where one party directly sent the share to the other party but two parties broadcast different values, is regarded as a new pair of disputed parties.

Key Distribution and Maintenance. In this step, we present the full protocol for key distribution and maintenance. At the end of the protocol, either a new pair of disputed parties is identified or all twisted degree-t sharings are valid with overwhelming probability.

The functionality is presented in Functionality 23.

Functionality 23: $\mathcal{F}_{\text{Key}}$

1. On receiving $(\texttt{disputed}, P_i, P_j)$, where $(P_i, P_j) \notin \mathcal{D}isp$ and at least one of P_i, P_j is corrupted, $\mathcal{F}_{\text{Key}}$ sends $(\texttt{disputed}, P_i, P_j)$ to all parties and halts. On receiving $(\texttt{corrupted}, P_i)$, where $P_i \notin \mathcal{C}orr$ and P_i is corrupted, $\mathcal{F}_{\text{Key}}$ sends $(\texttt{corrupted}, P_i)$ to all parties and halts.
2. For every honest party P_v, $\mathcal{F}_{\text{Key}}$ randomly generates twisted degree-t sharings $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$ for every party $P_i \notin \mathcal{D}isp_v \bigcup \{P_v\}$ such that the shares of $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$ held by parties in $\mathcal{D}isp_v$ are set to be 0.
3. For every honest party P_v, $\mathcal{F}_{\text{Key}}$ distributes the shares of $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$ to parties not in $\mathcal{D}isp_v \bigcup \{P_v, P_i\}$. In addition, $\mathcal{F}_{\text{Key}}$ sends $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$ and $\boldsymbol{\mu}_{v\to i}$ to P_v.
4. On receiving $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$, where P_v is corrupted and $(P_v, P_i) \notin \mathcal{D}isp$, such that all shares lie on a vector of polynomials $\boldsymbol{f}(\cdot)$ of degree at most t such that $\boldsymbol{f}(\alpha_i) = \boldsymbol{\mu}_{v\to i}$, $\boldsymbol{f}(0) = \mathbf{0}$ and $\boldsymbol{f}(\alpha_j) = \mathbf{0}$ for all parties $P_j \in \mathcal{D}isp_v$, $\mathcal{F}_{\text{Key}}$ distributes the shares of $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$ to parties not in $\mathcal{D}isp_v \bigcup \{P_v, P_i\}$.

Now we give an overview of the protocol KEY, which realizes the functionality $\mathcal{F}_{\text{Key}}$. In the beginning of KEY, each party P_v checks whether P_v has distributed twisted degree-t sharings of authentication keys. In the case that P_v has not distributed the sharings (e.g., KEY is invoked the first time) or the shares of some party which is disputed with P_v are not all 0 (e.g., a new pair of disputed parties including P_v was identified after P_v distributed the sharings), P_v invokes KEY-DISTRIBUTION(P_v, P_i) for every $P_i \notin \mathcal{D}isp_v$. In the case that P_v has already distributed the sharings of authentication keys and the shares of parties in $\mathcal{D}isp_v$ are all 0, P_v does nothing.

Then, all parties invoke CHECK-KEY to check the validness of all twisted degree-t sharings of the authentication keys. In the case that some party P_s broadcasts $(\texttt{fault}, v, i)$ at the end of CHECK-KEY, all parties invoke FL-KEY to identify a new pair of disputed parties.

The description of KEY appears in Protocol 24. Note that each party P_v only needs to re-distribute the sharings of authentication keys $O(n)$ times. Therefore, KEY-DISTRIBUTION will be invoked at most $O(n^3)$ times. Further more, KEY will be invoked $O(n^2)$ times (once per segment). Therefore, the overall communication complexity of KEY (in the whole protocol) is $O(\ell n^4 \phi + n^6 \kappa)$ bits plus $O(n^3 \phi + n^2 \kappa) \cdot \mathcal{BC}$.

Protocol 24: KEY

1. For every party $P_v \in \mathcal{P}_{\texttt{active}}$, P_v checks whether it has distributed the twisted degree-t sharings of authentication keys and the shares of parties in $\mathcal{D}isp_v$ are 0. If not, P_v invokes KEY-DISTRIBUTION(P_v, P_i) for all $P_i \notin \mathcal{D}isp_v$.
2. All parties invoke CHECK-KEY. If all parties broadcast $\texttt{ok}$, halt. Otherwise, suppose P_s broadcasts $(\texttt{fault}, v, i)$. All parties proceed to the next step.
3. All parties invoke FL-KEY to identify a new pair of disputed parties and update $\mathcal{D}isp, \mathcal{C}orr$.

Lemma 5. *Protocol* KEY *computes $\mathcal{F}_{Key}$ with unconditional security in the presence of honest majority.*

Proof. Note that $\mathcal{F}_{\text{Key}}$ generates and distributes twisted degree-t sharings of authentication keys on behalf of honest parties and directly distributes the sharings received from corrupted parties. The correctness of KEY is straightforward.

Consider the following construction of a simulator $\mathcal{S}$.

In the beginning, $\mathcal{S}$ invokes $\mathcal{F}_{\text{Key}}$ and receives the shares of corrupted parties from $\mathcal{F}_{\text{Key}}$. When P_v needs to generate and distribute new twisted degree-t sharings of authentication keys, there are two cases.

- If P_v is honest, $\mathcal{S}$ distributes the shares received from $\mathcal{F}_{\text{Key}}$ to corrupted parties.
- If P_v is corrupted, $\mathcal{S}$ receives the shares of honest parties. For the shares of each twisted degree-t sharing, $\mathcal{S}$ verifies whether these shares lie on a polynomial $f(\cdot)$ of degree at most t such that $f(0) = 0$ and for all $P_j \in \mathcal{D}isp_v$, $f(\alpha_j) = 0$.
  - If these shares satisfy the above requirement, $\mathcal{S}$ recovers the whole sharings and sends to $\mathcal{F}_{\text{Key}}$. Note that, for each sharing, $\mathcal{S}$ receives at least t shares from corrupted parties (If P_i is an honest party, $\mathcal{S}$ does not receive the share of P_i). Together with the constrain $f(0) = 0$, $\mathcal{S}$ can recover the whole sharing.
  - Otherwise, $\mathcal{S}$ does nothing.

Then, when checking the validity of all twisted degree-t sharings of the authentication keys, for every pair of parties $(P_v, P_i) \notin \mathcal{D}isp$ where P_v is honest, $\mathcal{S}$ sets the shares of $\lceil \mu^{(0)}_{v\to i} \rfloor^i_t$ held by parties in $\mathcal{D}isp_v$ to be 0 and distributes uniform elements in $\mathbb{K}$ to corrupted parties not in $\mathcal{D}isp_v$ as their shares of $\lceil \mu^{0}_{v\to i} \rfloor^i_t$ on behalf of P_v. $\mathcal{S}$ honestly follows the protocol CHALLENGE. For every pair of parties $(P_v, P_i) \notin \mathcal{D}isp$ where P_v is honest, $\mathcal{S}$ first computes the shares of $\lceil \sigma_{v\to i} \rfloor^i_t$ corrupted parties should hold. Then, $\mathcal{S}$ randomly samples a polynomial $f(\cdot)$ of degree at most t such that all shares of corrupted parties lie on it and $f(0) = 0$. For every honest party P_j, $\mathcal{S}$ sets the share of P_j to be $f(\alpha_j)$.

$\mathcal{S}$ behaves honestly in the remaining steps. When a new pair of disputed parties (P_i, P_j) is identified after FL-KEY, it is easy to see that at least one of P_i, P_j is corrupted. $\mathcal{S}$ sends $(\texttt{disputed}, P_i, P_j)$ to $\mathcal{F}_{\text{Key}}$.

Note that, the only difference between the real world execution and the ideal world execution is when a corrupted party P_v distributes invalid twisted degree-t sharings and all parties broadcast `ok` in the end of CHECK-KEY. In this case, $\mathcal{S}$ does not send the twisted degree-t sharings to $\mathcal{F}_{\text{Key}}$ and thus honest parties in the ideal world do not receive their shares, which is different from the real world. By Lemma 4, this scenario only happens with negligible probability.

5.4 Generating Authentication Tags

Recall that we have already verified the correctness of twisted sharings of authentication keys in KEY and the consistency of sharings distributed by each dealer in VERIFY-SHARING.

Computing Authentication Tags. In this part, we introduce a protocol to compute the authentication tags of the shares dealt by active parties.

Recall that $m = C/n^2$ is the size of each segment. We have further set $\ell = O(\frac{m}{t+1})$ and $q = \ell/e$. In Particular, q is the size of the authentication keys generated in KEY and ℓ is the number of shares we can authenticate each time.

For every pair of parties (P_v, P_i) and every dealer P_d, our goal is to compute an authentication tag for every ℓ shares sent from P_d to P_i using the authentication keys of P_v. We use $\boldsymbol{s}_i(d) \in \mathbb{F}^\ell$ to represent the vector of shares we want to authenticate. By mapping every e shares in $\mathbb{F}$ to one share in $\mathbb{K}$, $\boldsymbol{s}_i(d) \in \mathbb{F}^\ell$ is mapped to $\boldsymbol{\sigma}_i(d) \in \mathbb{K}^q$. For each batch, P_v randomly samples $\nu(d) \in \mathbb{K}$ and we want to compute the authentication tag $\tau(d) = \boldsymbol{\mu}_{v\to i} \odot \boldsymbol{\sigma}_i(d) + \nu(d)$ for P_i, where $\boldsymbol{\mu}_{v\to i}$ is the authentication keys generated in KEY and $\odot$ is the inner-product operation.

To compute the authentication tag, we need to secret-share every component used to compute the authentication tag. Note that $\boldsymbol{\mu}_{v\to i}$ has already been shared in KEY and $[\boldsymbol{\sigma}(d)]_t$ can be seen as the sharings

of $\boldsymbol{\sigma}_i(d)$ except that the secrets are hidden at position α_i. P_v randomly samples and distributes a twisted degree-$2t$ sharing $\lceil \nu(d) \rfloor_{2t}^i$. In addition P_d randomly samples and distributes a twisted degree-$2t$ sharing $\lceil o \rfloor_{2t}^i$ where $o = 0$ to protect the secrecy of $[\boldsymbol{\sigma}(d)]_t$.

Then we have

$$\lceil \tau(d) \rfloor_{2t}^i = \lceil \boldsymbol{\mu}_{v \to i} \rfloor_t^i \odot [\boldsymbol{\sigma}(d)]_t + \lceil \nu(d) \rfloor_{2t}^i + \lceil o \rfloor_{2t}^i.$$

Finally, all parties in $\mathcal{P}_{\texttt{active}}$ send their shares of $\lceil \tau(d) \rfloor_{2t}^i$ to P_i. Note that, each dealer P_d has set the shares held by parties in $\mathcal{C}orr \subseteq \mathcal{D}isp_d$ to be 0, which means that the shares of $\lceil \boldsymbol{\mu}_{v \to i} \rfloor_t^i, [\boldsymbol{\sigma}(d)]_t, \lceil \nu(d) \rfloor_{2t}^i, \lceil o \rfloor_{2t}^i$ held by parties in $\mathcal{C}orr$ are 0. Therefore, P_i can reconstruct $\tau(d)$ by using the shares of $\lceil \tau(d) \rfloor_{2t}^i$ held by parties in $\mathcal{P}_{\texttt{active}}$ and setting the shares held by parties in $\mathcal{C}orr$ to be 0.

The description of SingleTagComp appears in Protocol 25. The communication complexity of SingleTagComp is $O(n\kappa)$ bits.

Protocol 25: $\text{SingleTagComp}_{v,i,d}([\boldsymbol{\sigma}(d)]_t)$

1. P_v randomly samples $\nu \in \mathbb{K}$ and generates $\lceil \nu \rfloor_{2t}^i$ such that the shares of parties in $\mathcal{D}isp_v$ are set to be 0.
2. P_v distributes the shares of $\lceil \nu \rfloor_{2t}^i$ to parties not in $\mathcal{D}isp_v \bigcup \{P_v, P_i\}$.
3. P_d randomly generates $\lceil o \rfloor_{2t}^i$ such that $o = 0$ and the shares of parties in $\mathcal{D}isp_d$ are set to be 0.
4. P_d distributes the shares of $\lceil o \rfloor_{2t}^i$ to parties not in $\mathcal{D}isp_d \bigcup \{P_d, P_i\}$.
5. All parties compute $\lceil \tau \rfloor_{2t}^i = \lceil \boldsymbol{\mu}_{v \to i} \rfloor_t^i \odot [\boldsymbol{\sigma}(d)]_t + \lceil \nu \rfloor_{2t}^i + \lceil o \rfloor_{2t}^i$. Then every party $P_j \in \mathcal{P}_{\texttt{active}}$ sends its share to P_i (via the relay $P_{i \leftrightarrow j}$ if $(P_i, P_j) \in \mathcal{D}isp$).
6. P_i reconstructs τ by setting the shares of $\lceil \tau \rfloor_{2t}^i$ held by parties in $\mathcal{C}orr$ to be 0.

Checking the Correctness. Since we use a larger batch size, for every pair of parties $(P_v, P_i) \notin \mathcal{D}isp$, all batches of shares and authentication tags are verified at once. As a comparison, in [BSFO12], this is done for every three parties (P_v, P_i, P_d). For simplicity, we consider the case where each party deals $O(\frac{m}{t+1})$ sharings. It can be simply adapted to handle the case where only one party deals $O(m)$ sharings *without increasing the communication complexity.*

Let $w = O(1)$ denote the number of batches in each segment, i.e., each party deals $w \cdot \ell$ sharings. Let $\boldsymbol{\sigma}_i^{(k)}(d)$ denote the k-th batch of shares sent from P_d to P_i, $\nu^{(k)}(d)$ denote the key for the k-th batch, and $\tau^{(k)}(d)$ denote the tag for the k-th batch.

To check the correctness of all authentication tags. Each party will in additional distributes ℓ random degree-t sharings as random masks. These new sharings are only used in the checking phase and will be discarded after this step. Also, all parties need to make sure the generated sharings held by honest parties are consistent. The functionality is described in Functionality 26. The realization of $\mathcal{F}_{\text{BaseSharing}}$ can be found in [BSFO12]. The communication complexity of the realization in [BSFO12] is $O(\ell n^2 \phi + n^3 \kappa)$ bits plus $O(n\kappa) \cdot \mathcal{BC}$.

For every pair of parties $(P_v, P_i) \notin \mathcal{D}isp$, let $\boldsymbol{\sigma}_i^{(0)}(1), \boldsymbol{\sigma}_i^{(0)}(2), \ldots, \boldsymbol{\sigma}_i^{(0)}(n)$ denote the shares P_i received. Then P_v, P_i invoke $\text{SingleTagComp}_{v,i,d}([\boldsymbol{\sigma}^{(0)}(d)]_t)$ to compute the authentication tag $\tau^{(0)}(d)$ of $\boldsymbol{\sigma}_i^{(0)}(d)$ for every P_d.

Functionality 26: $\mathcal{F}_{\text{BaseSharing}}(\ell)$

1. On receiving $(\texttt{disputed}, P_i, P_j)$, where $(P_i, P_j) \notin \mathcal{D}isp$ and at least one of P_i, P_j is corrupted, $\mathcal{F}_{\text{BaseSharing}}$ sends $(\texttt{disputed}, P_i, P_j)$ to all parties and halts. On receiving $(\texttt{corrupted}, P_i)$, where $P_i \notin \mathcal{C}orr$ and P_i is corrupted, $\mathcal{F}_{\text{BaseSharing}}$ sends $(\texttt{corrupted}, P_i)$ to all parties and halts.
2. For every honest party P_i, $\mathcal{F}_{\text{BaseSharing}}$ randomly generates ℓ degree-t sharings $[s^{(1)}(i)]_t, \ldots, [s^{\ell}(i)]_t$ such that the shares held by parties in $\mathcal{D}isp_i$ are set to be 0.
3. For each corrupted party $P_i \notin \mathcal{C}orr$, $\mathcal{F}_{\text{BaseSharing}}$ receives $[s^{(1)}(i)]_t, \ldots, [s^{(\ell)}(i)]_t$ from P_i where for each $k \in [\ell]$, $[s^{(k)}(i)]_t$ is a consistent sharing such that the shares held by parties in $\mathcal{D}isp_i$ are 0.
4. For each party $P_i \in \mathcal{P}_{\texttt{active}}$, $\mathcal{F}_{\text{BaseSharing}}$ distributes the shares of $[s^{(1)}(i)]_t, \ldots, [s^{(\ell)}(i)]_t$ to parties not in $\mathcal{D}isp_i \bigcup \{P_i\}$.
5. For each honest party P_i, $\mathcal{F}_{\text{BaseSharing}}$ sends $[s^{(1)}(i)]_t, \ldots, [s^{(\ell)}(i)]_t$ to P_i.

Consider the following 3 polynomials:

$$\boldsymbol{F}(X) := \sum_{d=1}^{n} \left(\sum_{k=0}^{w} \boldsymbol{\sigma}_i^{(k)}(d) X^k \right) X^{(d-1)(w+1)}$$
$$\Gamma(X) := \sum_{d=1}^{n} \left(\sum_{k=0}^{w} \tau^{(k)}(d) X^k \right) X^{(d-1)(w+1)}$$
$$\Delta(X) := \sum_{d=1}^{n} \left(\sum_{k=0}^{w} \nu^{(k)}(d) X^k \right) X^{(d-1)(w+1)}$$

If all authentication tags are correct, we should have $\Gamma(X) = \boldsymbol{\mu}_{v \to i} \odot \boldsymbol{F}(X) + \Delta(X)$. However, if at least one authentication tag is incorrect, then the above equation holds on at most $n(w+1)$ points. Thus, by testing a random point $\lambda \in \mathbb{K}$, with overwhelming probability, the above equation does not hold.

The description of Check-Tag appears in Protocol 27. The communication complexity of Check-Tag is $O(\ell n^2 \phi + n^4 \kappa)$ bits plus $O(n\kappa) \cdot \mathcal{BC}$.

Lemma 6. *If all parties broadcast* `ok` *in the end of* Check-Tag*, then with overwhelming probability, all authentication tags are correct.*

Fault Localization. In the case that some party P_v broadcast $(\texttt{fault}, i)$ in the end of Check-Tag, we need to find out a new pair of disputed parties.

The first step is to figure out the dealer P_d of the shares where the authentication tags are incorrect.

For each $d \in \{1, \ldots, n\}$, let

$$\boldsymbol{\sigma}'_i(d) = \sum_{k=0}^{w} \boldsymbol{\sigma}_i^{(k)}(d) \lambda^k, \ \tau'(d) = \sum_{k=0}^{w} \tau^{(k)}(d) \lambda^k, \ \nu'(d) = \sum_{k=0}^{w} \nu^{(k)}(d) \lambda^k.$$

Then, we have $\boldsymbol{\sigma}'_i = \sum_{d=1}^{n} \boldsymbol{\sigma}'_i(d) \lambda^{(d-1)(w+1)}$, $\tau' = \sum_{d=1}^{n} \tau'(d) \lambda^{(d-1)(w+1)}$ and $\nu' = \sum_{d=1}^{n} \nu'(d) \lambda^{(d-1)(w+1)}$.

To find such P_d, P_i sends $(\boldsymbol{\sigma}'_i(1), \boldsymbol{\sigma}'_i(2), \ldots, \boldsymbol{\sigma}'_i(n))$ and $(\tau'(1), \tau'(2), \ldots, \tau'(n))$ to P_v. P_v then checks whether the following two equations hold:

$$\boldsymbol{\sigma}'_i = \sum_{d=1}^{n} \boldsymbol{\sigma}'_i(d) \lambda^{(d-1)(w+1)} \quad \tau' = \sum_{d=1}^{n} \tau'(d) \lambda^{(d-1)(w+1)}$$

Protocol 27: CHECK-TAG

1. All parties invoke $\mathcal{F}_{\text{BaseSharing}}(\ell)$ to generate $[\boldsymbol{\sigma}^{(0)}(1)]_t, \ldots, [\boldsymbol{\sigma}^{(0)}(n)]_t$.
2. For every $P_v, P_i, P_d \in \mathcal{P}_{\texttt{active}}$ such that $(P_v, P_i), (P_i, P_d) \notin \mathcal{D}isp$, all parties invoke $\text{SINGLETAGCOMP}_{v,i,d}([\boldsymbol{\sigma}^{(0)}(d)]_t)$. In the end, P_i obtains $\tau^{(0)}(d)$ and P_v obtains $\nu^{(0)}(d)$.
3. All parties invoke CHALLENGE to generate $\lambda \in \mathbb{K}$.
4. For every $(P_v, P_i) \notin \mathcal{D}isp$, P_i computes

$$\boldsymbol{\sigma}'_i = \sum_{d=1}^{n} \left(\sum_{k=0}^{w} \boldsymbol{\sigma}_i^{(k)}(d) \lambda^k \right) \lambda^{(d-1)(w+1)}$$
$$\tau' = \sum_{d=1}^{n} \left(\sum_{k=0}^{w} \tau^{(k)}(d) \lambda^k \right) \lambda^{(d-1)(w+1)}.$$

   P_v computes

$$\nu' = \sum_{d=1}^{n} \left(\sum_{k=0}^{w} \nu^{(k)}(d) \lambda^k \right) \lambda^{(d-1)(w+1)}.$$

   P_i sends $\boldsymbol{\sigma}'_i$ and τ' to P_v. P_v accepts if $\tau' = \boldsymbol{\mu}_{v \to i} \odot \boldsymbol{\sigma}'_i + \nu'$. Otherwise P_v rejects.
5. For every $P_v \in \mathcal{P}_{\texttt{active}}$, P_v broadcasts $\texttt{ok}$ if it accepts all verifications. Otherwise P_v broadcasts $(\texttt{fault}, i)$ where i is the smallest index such that P_v rejects the verification of P_i.

If not, (P_v, P_i) is a new pair of disputed parties. Otherwise, P_v finds P_d where $\tau'(d) \neq \boldsymbol{\mu}_{v \to i} \odot \boldsymbol{\sigma}'_i(d) + \nu'(d)$.

The description of FL-TAG appears in Protocol 28. The communication complexity of FL-TAG is $O(\ell n \phi + n\kappa)$ bits plus $O(\phi) \cdot \mathcal{BC}$.

Protocol 28: FL-TAG

1. Let P_v be the first party which broadcast $(\texttt{fault}, i)$.
2. For all $d \in \{1, 2, \ldots, n\}$, P_i computes $\boldsymbol{\sigma}'_i(d) = \sum_{k=0}^{w} \boldsymbol{\sigma}_i^{(k)}(d) \lambda^k$ and $\tau'(d) = \sum_{k=0}^{w} \tau^{(k)}(d) \lambda^k$. Then P_i sends $(\boldsymbol{\sigma}'_i(1), \ldots, \boldsymbol{\sigma}'_i(n))$ and $(\tau'(1), \ldots, \tau'(n))$ to P_v.
3. P_v checks whether the following two equation hold:

$$\boldsymbol{\sigma}'_i = \sum_{d=1}^{n} \boldsymbol{\sigma}'_i(d) \lambda^{(d-1)(w+1)} \quad \tau' = \sum_{d=1}^{n} \tau'(d) \lambda^{(d-1)(w+1)}$$

   - If not, P_v broadcasts $(\texttt{accuse}, \texttt{i})$ and all parties regard (P_v, P_i) as a new pair of disputed parties.
   - Otherwise, P_v computes $\nu'(d) = \sum_{k=0}^{w} \nu^{(k)}(d) \lambda^k$ for all $d \in \{1, 2, \ldots, n\}$. P_v then broadcasts $(\texttt{fault}, d)$ if $\tau'(d) \neq \boldsymbol{\mu}_{v \to i} \odot \boldsymbol{\sigma}'_i(d) + \nu'(d)$.

In the case that P_v broadcast $(\texttt{fault}, d)$, we need to check the messages sent by all parties when computing the authentication tags of shares distributed by P_d. Let $\lceil o^{(k)} \rfloor_{2t}^{i}$ be the twisted degree-$2t$ sharing of 0 when computing the authentication tag for the k-th batch of shares $\boldsymbol{\sigma}_i^{(k)}(d)$ for all $k \in \{0, 1, \ldots, w\}$.

All parties locally compute the following sharings:

$$\lceil \tau'(d) \rfloor^i_{2t} = \sum_{k=0}^{w} \lceil \tau^{(k)}(d) \rfloor^i_{2t} \lambda^k$$
$$[\boldsymbol{\sigma}'(d)]_t = \sum_{k=0}^{w} [\boldsymbol{\sigma}^{(k)}(d)]_t \lambda^k$$
$$\lceil \nu'(d) \rfloor^i_{2t} = \sum_{k=0}^{w} \lceil \nu^{(k)}(d) \rfloor^i_{2t} \lambda^k$$
$$\lceil o' \rfloor^i_{2t} = \sum_{k=0}^{w} \lceil o^{(k)} \rfloor^i_{2t} \lambda^k$$

Then, we should have $\lceil \tau'(d) \rfloor^i_{2t} = \lceil \boldsymbol{\mu}_{v \to i} \rfloor^i_t \odot [\boldsymbol{\sigma}'(d)]_t + \lceil \nu'(d) \rfloor^i_{2t} + \lceil o' \rfloor^i_{2t}$. To protect the secrecy of $\boldsymbol{\mu}_{v \to i}$, all parties need to send their shares of above sharings to P_v. In addition, P_d needs to send $[\boldsymbol{\sigma}'(d)]_t, \lceil o' \rfloor^i_{2t}$ to P_v and P_i needs to send $\lceil \tau'(d) \rfloor^i_{2t}$ to P_v. In this way, P_v is able to find a new pair of disputed parties.

The description of FL-SingleDealer appears in Protocol 29. The communication complexity of FL-SingleDealer is $O(\ell n \phi + n\kappa)$ bits plus $O(\kappa) \cdot \mathcal{BC}$.

Protocol 29: FL-SingleDealer(P_v, P_d)

1. All parties locally compute $\lceil \tau'(d) \rfloor^i_{2t}$, $[\boldsymbol{\sigma}'(d)]_t$ and $\lceil o' \rfloor^i_{2t}$. Then they send their shares to P_v (via relays if necessary).
2. P_d sends $[\boldsymbol{\sigma}'(d)]_t$ and $\lceil o' \rfloor^i_{2t}$ to P_v (via the relay $P_{v \leftrightarrow d}$ if $(P_v, P_d) \in \mathcal{D}isp$).
3. P_i sends $\lceil \tau'(d) \rfloor^i_{2t}$ to P_v.
4. P_v checks the following.
   - If P_v observes that some party P_j does not follow the protocol, P_v broadcast $(\texttt{accuse}, P_j)$. If $(P_v, P_j) \notin \mathcal{D}isp$, (P_v, P_j) is a new pair of disputed parties. Otherwise, the relay $P_{v \leftrightarrow j}$ checks the messages it helped transfer and broadcasts its opinion. If $P_{v \leftrightarrow j}$ agrees with P_v, $(P_{v \leftrightarrow j}, P_j)$ is a new pair of disputed parties. Otherwise, $(P_{v \leftrightarrow j}, P_v)$ is a new pair of disputed parties. (Note that if $(P_v, P_j) \in \mathcal{D}isp$, then the j-th share of $\lceil \tau'(d) \rfloor^i_{2t}$ should be the same as the j-th share of $\lceil o' \rfloor^i_{2t}$ since every j-th share of the sharings dealt by P_v is set to be 0.)
   - If P_v receives different values for some share of $\lceil \tau'(d) \rfloor^i_{2t}, [\boldsymbol{\sigma}'(d)]_t$ or $\lceil o' \rfloor^i_{2t}$, say the j-th share of $\lceil o' \rfloor^i_{2t}$ for concreteness, let o'_j and $\widetilde{o'_j}$ denote the shares received from P_d and P_j respectively.
     (a) P_v broadcasts $(\texttt{open}, j, o'_j, \widetilde{o'_j})$.
     (b) P_j, P_d and the relays $P_{v \leftrightarrow d}, P_{v \leftrightarrow j}$ broadcast the j-th shares of $\lceil o' \rfloor^i_{2t}$ they sent/received.
     (c) The pair of parties, where one party directly sends the share to the other party but two parties broadcast different values, is regarded as a new pair of disputed parties.

The Full Protocol. In this part, we introduce the protocol which realizes $\mathcal{F}_{\text{Tag}}$. The description of Tag appears in Protocol 30. The overall communication complexity of Tag (in the whole protocol) is $O(\ell n^4 \phi + n^6 \kappa)$ bits plus $O(n^3 \kappa) \cdot \mathcal{BC}$.

Lemma 7. *Protocol* Tag *computes $\mathcal{F}_{Tag}$ with unconditional security in the presence of honest majority.*

Proof. The correctness of Tag is straightforward. Consider the following construction of a simulator $\mathcal{S}$.

Protocol 30: TAG

1. All parties invoke $\mathcal{F}_{\text{Key}}$.
2. For every $P_d \in \mathcal{P}_{\texttt{active}}$, let $[\boldsymbol{\sigma}^{(1)}(d)]_t, \ldots, [\boldsymbol{\sigma}^{(w)}(d)]_t$ denote the sharings dealt by P_d. For every $P_v, P_i, P_d \in \mathcal{P}_{\texttt{active}}$ such that $(P_v, P_i), (P_i, P_d) \notin \mathcal{D}isp$, all parties invoke $\text{SINGLETAGCOMP}_{v,i,d}([\boldsymbol{\sigma}^{(k)}(d)]_t)$ for all $k \in \{1, 2, \ldots, w\}$.
3. All parties invoke CHECK-TAG. If all parties broadcast $\texttt{ok}$, all parties halt. Otherwise, suppose P_v broadcasts $(\texttt{fault}, i)$. All parties proceed to the next step.
4. All parties invoke FL-TAG. If a new pair of disputed parties is identified, all parties halt. Otherwise, suppose P_v broadcasts $(\texttt{fault}, d)$. All parties proceed to the next step.
5. All parties invoke FL-$\text{SINGLEDEALER}(P_v, P_d)$ to find a new pair of disputed parties and halt.

Simulating Step 1 of TAG. In Step 1, $\mathcal{S}$ simulates the functionality $\mathcal{F}_{\text{Key}}$.

1. $\mathcal{S}$ behaves honestly when receiving $(\texttt{disputed}, P_i, P_j)$ and $(\texttt{corrupted}, P_i)$.
2. For every pair of parties $(P_v, P_i) \notin \mathcal{D}isp$ such that P_v is honest, if P_i is corrupted, $\mathcal{S}$ set the shares of $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$ held by parties in $\mathcal{D}isp_v$ to be 0 and the shares held by other corrupted parties to be uniform field elements. If P_i is honest, $\mathcal{S}$ requests $\boldsymbol{\mu}_{v\to i}$ from $\mathcal{F}_{\text{Tag}}$ and samples a vector of random twisted degree-t sharings as $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$ such that the secret is $\boldsymbol{\mu}_{v\to i}$ and the shares held by parties in $\mathcal{D}isp_v$ are 0.
3. For every honest party P_v, $\mathcal{S}$ distributes the shares of $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$ to parties not in $\mathcal{D}isp_v \bigcup \{P_v, P_i\}$.
4. $\mathcal{S}$ faithfully checks the sharings received from corrupted parties and reconstructs the keys.

Simulating Step 2 of TAG. In Step 2, let $[\boldsymbol{\sigma}^{(1)}(d)]_t, \ldots, [\boldsymbol{\sigma}^{(w)}(d)]_t$ denote the sharings dealt by P_d. Note that if P_d is an honest party, then $\mathcal{S}$ learns the shares held by corrupted parties. If P_d is a corrupted party, then $\mathcal{S}$ learns the whole sharings and in particular, the each sharing dealt by P_d is consistent (since we have checked the consistency in VERIFY-SHARING before computing the authentication tags). For each corrupted party P_d, $\mathcal{S}$ sends (each of) $[\boldsymbol{\sigma}^{(1)}(d)]_t, [\boldsymbol{\sigma}^{(2)}(d)]_t, \ldots, [\boldsymbol{\sigma}^{(w)}(d)]_t$ to (each call of) $\mathcal{F}_{\text{Tag}}$.

Now, $\mathcal{S}$ needs to simulate the behaviors of honest parties in SINGLETAGCOMP. We use wide-tilde over a sharing to represent the sharing it should be (in contrast to the sharing received from all parties), i.e., replacing the shares received from corrupted parties by the shares corrupted parties should hold. Depending on whether P_v, P_i, P_d are honest or corrupted respectively, there are 8 cases.

- **Case 1:** If P_v, P_i, P_d are honest, $\mathcal{S}$ randomly generates and distributes $\lceil \nu^{(k)}(d) \rfloor_{2t}^i$ and $\lceil o^{(k)} \rfloor_{2t}^i$ on behalf of P_v, P_d respectively. After receiving the shares of $\lceil \tau^{(k)}(d) \rfloor_{2t}^i$ from corrupted parties, $\mathcal{S}$ computes the shares of $\lceil \tau^{(k)}(d) \rfloor_{2t}^i$ corrupted parties should hold and checks whether

  $$\lceil O \rfloor_{2t}^i = \lceil \tau^{(k)}(d) \rfloor_{2t}^i - \widetilde{\lceil \tau^{(k)}(d) \rfloor_{2t}^i}$$

  is a twisted degree-$2t$ sharing of 0. Note that the shares of $\lceil O \rfloor_{2t}^i$ held by honest parties are all 0. If true, $\mathcal{S}$ marks this invocation as $\texttt{accept}$. Otherwise $\mathcal{S}$ marks this invocation as $\texttt{reject}$.
- **Case 2:** If P_v, P_i are honest and P_d is corrupted, $\mathcal{S}$ randomly generates and distributes $\lceil \nu^{(k)}(d) \rfloor_{2t}^i$ on behalf of P_v. $\mathcal{S}$ receives from P_d the shares of $\lceil o^{(k)} \rfloor_{2t}^i$ held by honest parties. Let $\lceil \eta \rfloor_{2t}^i = \lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i \odot [\boldsymbol{\sigma}^{(k)}(d)]_t + \lceil \nu^{(k)}(d) \rfloor_{2t}^i$. $\mathcal{S}$ computes the shares of $\lceil \eta \rfloor_{2t}^i$ corrupted parties should hold. After receiving the shares of $\lceil \tau^{(k)}(d) \rfloor_{2t}^i$ from corrupted parties, $\mathcal{S}$ checks whether

  $$\lceil O \rfloor_{2t}^i = \lceil \tau^{(k)}(d) \rfloor_{2t}^i - \widetilde{\lceil \eta \rfloor_{2t}^i}$$

  is a twisted degree-$2t$ sharing of 0. Note that the shares of $\lceil O \rfloor_{2t}^i$ held by honest parties are the same as the shares of $\lceil o^k \rfloor_{2t}^i$ received from corrupted parties. If true, $\mathcal{S}$ marks this invocation as $\texttt{accept}$. Otherwise $\mathcal{S}$ marks this invocation as $\texttt{reject}$.

- **Case 3:** If P_v, P_d are honest and P_i is corrupted, $\mathcal{S}$ randomly generates and distributes $\lceil o^{(k)} \rfloor_{2t}^i$ on behalf of P_d. $\mathcal{S}$ receives $\tau^{(k)}(d)$ from $\mathcal{F}_{\text{Tag}}$. $\mathcal{S}$ randomly samples a twisted degree-$2t$ sharing $\lceil \eta \rfloor_{2t}^i$ such that the shares held by parties in $\mathcal{D}isp_v$ are set to be 0 and $\eta = \tau^{(k)}(d)$. Set $\lceil \nu^{(k)}(d) \rfloor_{2t}^i = \lceil \eta \rfloor_{2t}^i - \lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i \odot [\boldsymbol{\sigma}^{(k)}(d)]_t$. Note that, $\mathcal{S}$ is able to compute the shares of $\lceil \nu^{(k)}(d) \rfloor_{2t}^i$ held by corrupted parties. $\mathcal{S}$ distributes $\lceil \nu^{(k)}(d) \rfloor_{2t}^i$ on behalf of P_v. Finally, $\mathcal{S}$ sets $\lceil \tau^{(k)}(d) \rfloor_{2t}^i = \lceil \eta \rfloor_{2t}^i + \lceil o^{(k)} \rfloor_{2t}^i$ and sends the shares of honest parties to P_i accordingly.
- **Case 4:** If P_v is honest and P_i, P_d are corrupted, $\mathcal{S}$ receives $\tau^{(k)}(d)$ from $\mathcal{F}_{\text{Tag}}$. $\mathcal{S}$ samples $\lceil \eta \rfloor_{2t}^i$ and computes the shares of $\lceil \nu^{(k)}(d) \rfloor_{2t}^i$ held by corrupted parties in the same way as that in Case 3. Then $\mathcal{S}$ distributes $\lceil \nu^{(k)}(d) \rfloor_{2t}^i$ on behalf of P_v. Finally, $\mathcal{S}$ sets $\lceil \tau^{(k)}(d) \rfloor_{2t}^i = \lceil \eta \rfloor_{2t}^i + \lceil o^{(k)} \rfloor_{2t}^i$ and sends the shares of honest parties to P_i accordingly.
- **Case 5:** If P_i, P_d are honest and P_v is corrupted, $\mathcal{S}$ randomly generates and distributes $\lceil o^{(k)} \rfloor_{2t}^i$ on behalf of P_d. $\mathcal{S}$ receives the shares of $\lceil \nu^{(k)}(d) \rfloor_{2t}^i$ held by honest parties. After receiving the shares of $\lceil \tau^{(k)}(d) \rfloor_{2t}^i$ held by corrupted parties, $\mathcal{S}$ computes the shares of $\lceil \nu^{(k)}(d) \rfloor_{2t}^i = \lceil \tau^{(k)}(d) \rfloor_{2t}^i - \lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i \odot [\boldsymbol{\sigma}^{(k)}(d)]_t - \lceil o^{(k)} \rfloor_{2t}^i$ held by corrupted parties. Then $\mathcal{S}$ can reconstruct $\lceil \nu^{(k)}(d) \rfloor_{2t}^i$ and compute $\nu^{(k)}(d)$. Finally $\mathcal{S}$ sends $(\boldsymbol{\mu}_{v\to i}, \nu^{(k)}(d))$ to $\mathcal{F}_{\text{Tag}}$.
- **Case 6:** If P_i is honest and P_v, P_d are corrupted, $\mathcal{S}$ receives the shares of $\lceil \nu^{(k)}(d) \rfloor_{2t}^i$ and $\lceil o^{(k)} \rfloor_{2t}^i$ held by honest parties. Then $\mathcal{S}$ computes the shares of $\lceil \nu^{(k)}(d) \rfloor_{2t}^i + \lceil o^{(k)} \rfloor_{2t}^i$ held by honest parties. After receiving the shares of $\lceil \tau^{(k)}(d) \rfloor_{2t}^i$ held by corrupted parties, $\mathcal{S}$ computes the shares of $\lceil \nu^{(k)}(d) \rfloor_{2t}^i + \lceil o^{(k)} \rfloor_{2t}^i = \lceil \tau^{(k)}(d) \rfloor_{2t}^i - \lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i \odot [\boldsymbol{\sigma}^{(k)}(d)]_t$ held by corrupted parties. Then $\mathcal{S}$ can reconstruct $\lceil \nu^{(k)}(d) \rfloor_{2t}^i + \lceil o^{(k)} \rfloor_{2t}^i$ and compute $\nu^{(k)}(d) + o^{(k)} = \nu^{(k)}(d)$. Finally $\mathcal{S}$ sends $(\boldsymbol{\mu}_{v\to i}, \nu^{(k)}(d))$ to $\mathcal{F}_{\text{Tag}}$.
- **Case 7:** If P_d is honest and P_v, P_i are corrupted, $\mathcal{S}$ first computes $\gamma = \boldsymbol{\mu}_{v\to i} \odot \boldsymbol{\sigma}_i^{(k)}(d)$. Note that, $\mathcal{S}$ has reconstructed $\boldsymbol{\mu}_{v\to i}$ when simulating $\mathcal{F}_{\text{Key}}$. And $\boldsymbol{\sigma}_i^k(d)$ are the shares P_d sent to P_i. Then $\mathcal{S}$ randomly samples a twisted degree-$2t$ sharing $\lceil \gamma \rfloor_{2t}^i$ such that the shares held by parties in $\mathcal{D}isp_d$ are set to be 0. Set $\lceil o^{(k)} \rfloor_{2t}^i = \lceil \gamma \rfloor_{2t}^i - \lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i \odot [\boldsymbol{\sigma}^{(k)}(d)]_t$. Note that $\mathcal{S}$ is able to compute the shares of $\lceil o^{(k)} \rfloor_{2t}^i$ held by corrupted parties. $\mathcal{S}$ distributes $\lceil o^{(k)} \rfloor_{2t}^i$ on behalf of P_d. Finally, $\mathcal{S}$ sets $\lceil \tau^{(k)}(d) \rfloor_{2t}^i = \lceil \gamma \rfloor_{2t}^i + \lceil \nu^{(k)}(d) \rfloor_{2t}^i$ and sends the shares of honest parties to P_i accordingly. $\mathcal{S}$ randomly samples $\nu^{(k)}(d)$ and sends $(\boldsymbol{\mu}_{v\to i}, \nu^{(k)}(d))$ to $\mathcal{F}_{\text{Tag}}$. $\mathcal{S}$ ignores the tag received from $\mathcal{F}_{\text{Tag}}$.
- **Case 8:** If P_v, P_i, P_d are corrupted, $\mathcal{S}$ behaves honestly. $\mathcal{S}$ randomly samples $\nu^{(k)}(d)$ and sends $(\boldsymbol{\mu}_{v\to i}, \nu^{(k)}(d))$ to $\mathcal{F}_{\text{Tag}}$. $\mathcal{S}$ ignores the tag received from $\mathcal{F}_{\text{Tag}}$.

Simulating Step 3 of Tag. In Step 3, $\mathcal{S}$ needs to simulate the behaviors of honest parties when executing Check-Tag.

1. $\mathcal{S}$ first simulates $\mathcal{F}_{\text{BaseSharing}}$ as following. $\mathcal{S}$ behaves honestly when receiving $(\texttt{disputed}, P_i, P_j)$ and $(\texttt{corrupted}, P_i)$. For each honest party P_d, $\mathcal{S}$ sets the shares of $[\boldsymbol{\sigma}^{(0)}(d)]_t$ held by parties in $\mathcal{D}isp_d$ to be 0 and the shares held by other corrupted parties to be random field elements. For each corrupted party P_d, $\mathcal{S}$ receives the sharings $[\boldsymbol{\sigma}^{(0)}(d)]_t$ from adversaries. $\mathcal{S}$ then faithfully follows the rest of steps in $\mathcal{F}_{\text{BaseSharing}}$.
2. $\mathcal{S}$ simulates the invocations of $\text{SingleTagComp}_{v,i,d}([\boldsymbol{\sigma}^{(0)}(d)]_t)$ in the same way as described above.
3. $\mathcal{S}$ behaves honestly when generating $\lambda \in \mathbb{K}$.
4. For each honest party P_d, $\mathcal{S}$ randomly samples $[\boldsymbol{\sigma}'(d)]_t$ based on the shares held by corrupted parties. For each corrupted party P_d, $\mathcal{S}$ faithfully computes $[\boldsymbol{\sigma}'(d)]_t = \sum_{k=0}^{w} [\boldsymbol{\sigma}^{(k)}(d)]_t \lambda^k$. Depending on whether P_v, P_i are honest or corrupted respectively, there are 4 cases.
   - **Case 1:** If P_v, P_i are honest, $\mathcal{S}$ checks whether $\mathcal{S}$ marked some invocation $\text{SingleTagComp}_{v,i,d}([\boldsymbol{\sigma}^{(k)}(d)]_t)$ as `reject`. If true, $\mathcal{S}$ marks that P_v rejects the verification of P_i.
   - **Case 2:** If P_v is honest and P_i is corrupted, $\mathcal{S}$ first checks whether the shares $\boldsymbol{\sigma}_i'$ are correct. Note that, in this case, $\mathcal{S}$ received all the tags from $\mathcal{F}_{\text{Tag}}$ (see **Case 3** and **Case 4** in Step 2). Therefore, $\mathcal{S}$ directly computes τ' and check whether it is the same as that received from P_i. If either of these two checks fails, $\mathcal{S}$ marks that P_v rejects the verification of P_i.
   - **Case 3:** If P_i is honest and P_v is corrupted, $\mathcal{S}$ computes

$$\boldsymbol{\sigma}_i' = \sum_{d=1}^{n} \boldsymbol{\sigma}_i'(d) \lambda^{(d-1)(w+1)}.$$

Note that, in this case, $\{\nu^{(k)}(d)\}_{d=1}^n$ have been reconstructed by $\mathcal{S}$ (see **Case 5** and **Case 6** in Step 2). $\mathcal{S}$ computes

$$\nu' = \sum_{d=1}^{n} \left(\sum_{k=0}^{w} \nu^{(k)}(d)\lambda^k \right) \lambda^{(d-1)(w+1)}.$$

Recall that $\mathcal{S}$ has recovered $\boldsymbol{\mu}_{v\to i}$ when simulating $\mathcal{F}_{\text{Key}}$. Finally $\mathcal{S}$ sends $\boldsymbol{\sigma}'_i$ and $\tau' = \boldsymbol{\mu}_{v\to i} \odot \boldsymbol{\sigma}'_i + \nu'$ to P_v.

- **Case 4:** If P_v, P_i are corrupted, $\mathcal{S}$ does nothing.

5. $\mathcal{S}$ follows the protocol for all honest parties.

Simulating Step 4 of Tag. In Step 4, $\mathcal{S}$ needs to simulate the behaviors of honest parties when executing FL-Tag. Depending on whether P_v, P_i are honest or corrupted respectively, there are 4 cases.

- **Case 1:** If P_v, P_i are honest, $\mathcal{S}$ broadcasts $(\texttt{fault}, d)$ where d is the smallest index such that the invocation $\text{SingleTagComp}_{v,i,d}([\boldsymbol{\sigma}^{(k)}(d)]_t)$ is marked as `reject` for some $k \in \{0, 1, \ldots, w\}$.
- **Case 2:** If P_v is honest and P_i is corrupted, $\mathcal{S}$ first checks whether $\boldsymbol{\sigma}'_i = \sum_{d=1}^n \boldsymbol{\sigma}'_i(d)\lambda^{(d-1)(w+1)}$ and $\tau' = \sum_{d=1}^n \tau'(d)\lambda^{(d-1)(w+1)}$. If not, $\mathcal{S}$ broadcasts $(\texttt{accuse}, i)$ on behalf of P_v. Otherwise, $\mathcal{S}$ directly computes $\{\tau'(d)\}_{d=1}^n$ and broadcasts $(\texttt{fault}, d)$ where d is the smallest index such that $\tau'(d)$ received from P_i is incorrect. ($\mathcal{S}$ received all tags from $\mathcal{F}_{\text{Tag}}$ in this case.)
- **Case 3:** If P_i is honest and P_v is corrupted, $\mathcal{S}$ computes

$$\nu'(d) = \sum_{k=0}^{w} \nu^{(k)}(d)\lambda^k.$$

for all $d \in \{1, 2, \ldots, n\}$. Then $\mathcal{S}$ computes $\tau'(d) = \boldsymbol{\mu}_{v\to i} \odot \boldsymbol{\sigma}'_i(d) + \nu'(d)$ for all $d \in \{1, \ldots, n\}$. Finally, $\mathcal{S}$ sends $(\boldsymbol{\sigma}'_i(1), \ldots, \boldsymbol{\sigma}'_i(n))$ and $(\tau'(1), \ldots, \tau'(n))$ to P_v.
- **Case 4:** If P_v, P_i are corrupted, $\mathcal{S}$ does nothing.

Simulating Step 5 of Tag. In Step 5, $\mathcal{S}$ needs to simulate the behaviors of honest parties when executing FL-SingleDealer. Depending on whether P_v is honest or not, there are two cases.

- **Case 1:** If P_v is honest, then $\mathcal{S}$ can simply follow the protocol. This is because P_v is the only recipient of all shares and sharings, i.e., $\mathcal{S}$ does not need to send any message to corrupted parties. Although $\mathcal{S}$ does not know the whole sharings $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i$, $\mathcal{S}$ knows the shares held by corrupted parties, which is enough to check whether corrupted parties followed the protocol.
- **Case 2:** If P_v is corrupted, then $\mathcal{S}$ only needs to prepare the messages honest parties need to send to corrupted parties. Note that, $\mathcal{S}$ knows the shares of $\lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i, \lceil \nu'(d) \rfloor_{2t}^i$ held by honest parties. Furthermore, $\mathcal{S}$ has known $[\boldsymbol{\sigma}'(d)]_t$, which is computed when simulating Check-Tag. In the case that P_d is corrupted, $\mathcal{S}$ knows the shares of $\lceil o' \rfloor_{2t}^i$ held by honest parties. In the case that P_d is honest, if P_i is honest, then it corresponds to **Case 5** when simulating $\text{SingleTagComp}_{v,i,d}$ and $\{\lceil o^{(k)} \rfloor_{2t}^i\}_{k=0}^w$ are explicitly generated. Otherwise, it corresponds to **Case 7** when simulating $\text{SingleTagComp}_{v,i,d}([\boldsymbol{\sigma}^{(k)}(d)]_t)$. Recall that, $\mathcal{S}$ explicitly generated $\lceil \gamma^{(k)} \rfloor_{2t}^i$ and $\lceil o^{(k)} \rfloor_{2t}^i = \lceil \gamma^{(k)} \rfloor_{2t}^i - \lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i \odot [\boldsymbol{\sigma}^{(k)}(d)]_t$. We have

$$\begin{aligned}
\lceil o' \rfloor_{2t}^i &= \sum_{k=0}^{w} \lceil o^{(k)} \rfloor_{2t}^i \lambda^k \\
&= \sum_{k=0}^{w} (\lceil \gamma^{(k)} \rfloor_{2t}^i - \lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i \odot [\boldsymbol{\sigma}^{(k)}(d)]_t)\lambda^k \\
&= \sum_{k=0}^{w} \lceil \gamma^{(k)} \rfloor_{2t}^i \lambda^k - \lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i \odot \sum_{k=0}^{w} [\boldsymbol{\sigma}^{(k)}(d)]_t \lambda^k \\
&= \sum_{k=0}^{w} \lceil \gamma^{(k)} \rfloor_{2t}^i \lambda^k - \lceil \boldsymbol{\mu}_{v\to i} \rfloor_t^i \odot [\boldsymbol{\sigma}'(d)]_t.
\end{aligned}$$

From the last equation, $\mathcal{S}$ can compute the shares of $\lceil o' \rfloor_{2t}^i$ held by honest parties. Furthermore, $\mathcal{S}$ can compute the shares of $\lceil \tau'(d) \rfloor_{2t}^i$ held by honest parties using the equation

$$\lceil \tau'(d) \rfloor_{2t}^i = \lceil \boldsymbol{\mu}_{v \to i} \rfloor_t^i \odot [\boldsymbol{\sigma}'(d)]_t + \lceil \nu'(d) \rfloor_{2t}^i + \lceil o' \rfloor_{2t}^i.$$

After preparing the above shares held by honest parties, $\mathcal{S}$ follows the rest of the protocol.

This finishes the description of the simulator $\mathcal{S}$. We claim that, with overwhelming probability, the view of corrupted parties in the ideal world is identical to that in the real world. We check it step by step.

In Step 1, it is clear that $\mathcal{S}$ perfectly simulates $\mathcal{F}_{\text{Key}}$.

In Step 2, we show that the distribution of messages sent from $\mathcal{S}$ to corrupted parties is identical to the distribution of messages sent from honest parties to corrupted parties in the real world.

- For **Case 1** and **Case 2**, $\mathcal{S}$ faithfully follows the protocol to generates the shares of honest parties.
- For **Case 3** and **Case 4**, $\nu^{(k)}(d)$ is a uniform element in $\mathbb{K}$ sampled by honest party P_v, which is identical to the ideal world. Furthermore, in the real world, P_v randomly samples $\lceil \nu^{(k)}(d) \rfloor_{2t}^i$ such that the shares held by parties in $\mathcal{D}isp_v$ are set to be 0. Since the shares of $\lceil \boldsymbol{\mu}_{v \to i} \rfloor_t^i$ held by parties in $\mathcal{D}isp_v$ are 0, $\lceil \eta \rfloor_{2t}^i = \lceil \boldsymbol{\mu}_{v \to i} \rfloor_t^i \odot [\boldsymbol{\sigma}^{(k)}(d)]_t + \lceil \nu^{(k)}(d) \rfloor_{2t}^i$ is a random twisted degree-$2t$ sharing such that $\eta = \tau^{(k)}(d)$ and the shares held by parties in $\mathcal{D}isp_v$ are 0. Note that the distribution of $\lceil \eta \rfloor_{2t}^i$ generated by $\mathcal{S}$ is identical to that in the real world. Therefore, the distribution of $\lceil \nu^{(k)}(d) \rfloor_{2t}^i$ generated by $\mathcal{S}$ is identical to the real world.
- For **Case 5** and **Case 6**, $\mathcal{S}$ faithfully follows the protocol to generates the shares of honest parties.
- For **Case 7**, P_d randomly samples $\lceil o^{(k)} \rfloor_{2t}^i$ such that the shares held by parties in $\mathcal{D}isp_d$ are set to be 0 in real world. Since the shares of $[\boldsymbol{\sigma}^{(k)}(d)]_t$ held by parties in $\mathcal{D}isp_d$ are also 0, $\lceil \gamma \rfloor_{2t}^i = \lceil \boldsymbol{\mu}_{v \to i} \rfloor_t^i \odot [\boldsymbol{\sigma}^{(k)}(d)]_t + \lceil o^{(k)} \rfloor_{2t}^i$ is a random twisted degree-$2t$ sharing such that $\gamma = \boldsymbol{\mu}_{v \to i} \odot \boldsymbol{\sigma}^{(k)}(d)$ and the shares held by parties in $\mathcal{D}isp_d$ are 0. Note that the distribution of $\lceil \gamma \rfloor_{2t}^i$ generated by $\mathcal{S}$ is identical to that in the real world. Therefore, the distribution of $\lceil o^{(k)} \rfloor_{2t}^i$ generated by $\mathcal{S}$ is identical to the real world.

In Step 3, it is clear that $\mathcal{S}$ perfectly simulates $\mathcal{F}_{\text{BaseSharing}}$. For all honest party P_d, since $[\boldsymbol{\sigma}^{(0)}(d)]_t$ are random degree-t sharings such that the shares held by parties in $\mathcal{D}isp_d$ are 0, $[\boldsymbol{\sigma}'(d)]_t = \sum_{k=0}^{w} [\boldsymbol{\sigma}^{(k)}(d)]_t \lambda^k$ are random degree-t sharings such that the shares held by parties in $\mathcal{D}isp_d$ are 0. Note that, the distribution of $[\boldsymbol{\sigma}'(d)]_t$ generated by $\mathcal{S}$ is identical to that in the real world. In Step 5 of Check-Tag, we show that $\mathcal{S}$ perfectly simulates honest parties in the real world with overwhelming probability.

- For **Case 1**, note that in **Case 1** and **Case 2** when simulating SingleTagComp, $\mathcal{S}$ has checked whether honest parties received the correct tags. In the ideal world, P_v rejects the verification of P_i if at least one tag is incorrect. According to Lemma 6, with overwhelming probability, P_v rejects the verification of P_i if at least one tag is incorrect.
- For **Case 2**, $\mathcal{S}$ directly checks whether $\boldsymbol{\sigma}_i'$ and the authentication tag are correct. The only difference between the real world and the ideal world is that, if P_i provided an incorrect $\boldsymbol{\sigma}_i'$ and a valid authentication tag, P_v will accept the verification of P_i in the real world while it will reject the verification in the ideal world. However, since the authentication keys $\boldsymbol{\mu}_{v \to i}$ are uniformly distributed given the shares of corrupted parties, it happens with negligible probability.
- For **Case 3**, $\mathcal{S}$ has recovered $\nu^{(k)}(d)$ chosen by corrupted party P_d in **Case 5** and **Case 6** when simulating SingleTagComp. Therefore, $\mathcal{S}$ simulates honest parties perfectly.

In Step 4, $\mathcal{S}$ perfectly simulates honest parties in **Case 1** and **Case 3**. In **Case 2**, the only difference is that, if P_i provided an incorrect $\boldsymbol{\sigma}_i'(d')$ and a valid authentication tag for some $d' \in \{1, 2, \ldots, n\}$, P_v may identify a different d from that in the real world. However, since the authentication keys $\boldsymbol{\mu}_{v \to i}$ are uniformly distributed given the shares of corrupted parties, it happens with negligible probability.

In Step 5, all messages sent by $\mathcal{S}$ can be computed from the shares $\mathcal{S}$ learned before. Therefore, in the case that $\mathcal{S}$ perfectly simulate honest parties in previous steps, $\mathcal{S}$ also perfectly simulate honest parties in this step.

Efficiency Analysis of Tag. Note that the overall communication complexity of TAG (in the whole protocol) is $O(\ell n^4\phi + n^6\kappa)$ bits plus $O(n^3\kappa)\cdot\mathcal{BC}$. Recall that $\ell = O(\frac{m}{t+1})$. For any constant $\epsilon > 0$, by setting $\ell = \frac{\epsilon m}{t+1}$, the overall communication complexity of TAG is only $O(\epsilon n\phi)$ bits per gate.

5.5 Analyze Sharing

In this part, we introduce the whole protocol ANALYZE-SHARING to handle an inconsistent sharing $[x]_t$.

Recall that each sharing $[x]_t$ which needs to be examined can be decomposed into the following form:

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

where $[x(i)]_t$ is a linear combination of the sharings dealt by P_i.

Localize A Suspect Dealer. Recall that when all parties want to examine some sharing $[x]_t$, each party has broadcast its share of $[x]_t$ and the sharing $[x]_t$ is inconsistent. The description of LOCALIZE appears in Protocol 31. The communication complexity of LOCALIZE is $O(n^2\kappa)$ bits plus $O(n\kappa)\cdot\mathcal{BC}$.

Protocol 31: LOCALIZE

All parties have broadcast their shares of $[x]_t$ and the sharing $[x]_t$ is inconsistent. Let the decomposition of $[x]_t$ be $[x]_t = \sum_{i=1}^{n}[x(i)]_t$. Recall that $P_{\texttt{king}}$ is the special party all parties agree on in the beginning of each segment.

1. All parties prepare a random degree-t sharing $[r]_t$ by invoking RAND in $\mathbb{K}$. Let the decomposition of $[r]_t$ be $[r]_t = \sum_{i=1}^{n}[r(i)]_t$, where $[r(i)]_t$ is dealt by P_i.
2. All parties broadcast their shares of $[r]_t$.
   - If $[r]_t$ is inconsistent, set $[z]_t := [r]_t$ and $[z(i)]_t := [r(i)]_t$ for all $i \in [n]$.
   - Otherwise, set $[z]_t := [x]_t + [r]_t$ and $[z(i)]_t = [x(i)]_t + [r(i)]_t$ for all $i \in [n]$.
3. All parties send their shares $\{[z(i)]_t\}_{i=1}^{n}$ to $P_{\texttt{king}}$ (via relays if necessary).
4. $P_{\texttt{king}}$ checks the following.
   - If for some party $P_j \in \mathcal{P}_{\texttt{active}}$, the summation of its shares of $\{[z(i)]_t\}_{i=1}^{n}$ does not equal to the share of $[z]_t$ broadcast by P_j. $P_{\texttt{king}}$ broadcasts $(\texttt{accuse}, P_j)$. If $(P_{\texttt{king}}, P_j) \notin \mathcal{D}isp$, $(P_{\texttt{king}}, P_j)$ is a new pair of disputed parties. Otherwise, the relay $P_{\texttt{king}\leftrightarrow j}$ broadcasts its opinion. If $P_{\texttt{king}\leftrightarrow j}$ agrees with $P_{\texttt{king}}$, $(P_{\texttt{king}\leftrightarrow j}, P_j)$ is a new pair of disputed parties. Otherwise, $(P_{\texttt{king}\leftrightarrow j}, P_{\texttt{king}})$ is a new pair of disputed parties.
   - Otherwise, $P_{\texttt{king}}$ broadcasts $(\texttt{inconsistent}, i)$ to indicate that $[z(i)]_t$ it received is inconsistent.
5. All parties broadcast their shares of $[z(i)]_t$. If $[z(i)]_t$ is consistent, $P_{\texttt{king}}$ broadcasts $(\texttt{accuse}, P_j)$ to indicate that P_j didn't send the correct share to $P_{\texttt{king}}$.
   - If $(P_{\texttt{king}}, P_j) \notin \mathcal{D}isp$, $(P_{\texttt{king}}, P_j)$ is a new pair of disputed parties.
   - Otherwise, the relay $P_{\texttt{king}\leftrightarrow j}$ broadcasts its opinion. If $P_{\texttt{king}\leftrightarrow j}$ agrees with $P_{\texttt{king}}$, $(P_{\texttt{king}\leftrightarrow j}, P_j)$ is a new pair of disputed parties. Otherwise, $(P_{\texttt{king}\leftrightarrow j}, P_{\texttt{king}})$ is a new pair of disputed parties.

   Otherwise, all parties take P_i as output.

Handling an Active Dealer. If $P_i \notin \mathcal{C}orr$, then P_i checks the sharing $[z(i)]_t$ and identify the party P_j who broadcast a wrong share. If $(P_i, P_j) \notin \mathcal{D}isp$, then all parties take (P_i, P_j) as a new pair of disputed parties and halt. Otherwise, all the sharings dealt by P_i in this segment should satisfy that the shares held by P_j are set to be 0. Therefore, the non-zero shares P_i sent to P_j were all generated in previous segments, which have been authenticated.

P_i and P_j will run a 3-round search to locate a problematic batch of shares. Recall that the number of segments is $O(n^2)$.

- All previous segments are partitioned into n equal parts. For each part, P_j broadcasts the linear combination of the shares sent by P_i using the same coefficients as those of the j-th share of $[z(i)]_t$. If the summation of the values broadcast by P_j does not match the j-th share of $[z(i)]_t$, all parties regard P_j as a new corrupted party. Otherwise, P_i broadcasts an index to point out the problematic part.
- For each segment of the problematic part pointed out by P_i, P_j broadcasts the linear combination of the shares sent by P_i using the same coefficients as those of the j-th share of $[z(i)]_t$. If the summation of the values broadcast by P_j does not match the value P_j broadcast for this part, all parties regard P_j as a new corrupted party. Otherwise, P_i broadcasts an index to point out the problematic segment.
- For each batch of the problematic segment pointed out by P_i, P_j broadcasts the linear combination of the shares sent by P_i using the same coefficients as those of the j-th share of $[z(i)]_t$. If the summation of the values broadcast by P_j does not match the value P_j broadcast for this segment, all parties regard P_j as a new corrupted party. Otherwise, P_i broadcasts an index to point out the problematic batch.

Then P_j sends the shares in this batch and the associated authentication tag to each other party P_v. P_v verifies the authentication tag and checks whether the linear combination of these shares using the same coefficients as those of the j-th share of $[z(i)]_t$ equals to the value P_j broadcast for this batch. If at least $t+1$ parties accept the shares, P_i is regarded as a corrupted party. Otherwise, P_j is regarded as a corrupted party.

The description of Active-Dealer appears in Protocol 32. The communication complexity of Active-Dealer is $O(\ell n\phi + n\kappa)$ bits plus $O(n\kappa) \cdot \mathcal{BC}$. Here ℓ is the size of each batch of shares.

Protocol 32: Active-Dealer

1. P_i broadcasts $(\texttt{accuse}, j)$ to indicate that the j-th share of $[z(i)]_t$ broadcast by P_j is incorrect. If $(P_i, P_j) \notin \mathcal{D}isp$, then (P_i, P_j) is a new pair of disputed parties. Otherwise, continue to the next step.
2. P_i, P_j run a 3-round search to locate one batch of shares which P_i, P_j do not agree on.
3. For every party $P_v \in \mathcal{P}_{\texttt{active}}$, P_j send this batch of shares together with the corresponding authentication tag to P_v. P_v checks the validity of the authentication tag and whether the linear combination of the shares in this batch using the same coefficients as those of the j-th share of $[z(i)]_t$ is the same as the value P_j broadcast for this batch. If both checks pass, P_v broadcasts `accept`. Otherwise, P_v broadcasts `reject`.
4. If majority broadcast `accept`, P_i is identified as a corrupted party. Otherwise, P_j is identified as a corrupted party.

Handling a Corrupted Dealer. If $P_i \in \mathcal{C}orr$, P_i did not participate in the computation of this segment. Therefore, the non-zero sharings dealt by P_i were all generated in previous segments, which have been authenticated.

All parties run a 3-round search to locate a problematic batch of shares using a similar way to that in Active-Dealer. The only difference is that each party now needs to broadcast its shares. All parties will use the index of the first inconsistent sharing as the problematic part, segment or batch. Then, each party P_j asks other parties to verify its shares in the same way as that in Active-Dealer.

The description of Corrupted-Dealer appears in Protocol 33. The communication complexity of Corrupted-Dealer is $O(\ell n^2\phi + n^2\kappa)$ bits plus $O(n^2\kappa) \cdot \mathcal{BC}$. Here ℓ is the size of each batch of shares.

Protocol 33: Corrupted-Dealer

1. All parties in $\mathcal{P}_{\texttt{active}}$ run a 3-round search to locate one batch of sharings where at least one sharing is inconsistent.
2. For every two parties $P_j, P_v \in \mathcal{P}_{\texttt{active}}$, P_j sends this batch of shares together with the corresponding authentication tag to P_v. P_v checks the validity of the authentication tag and whether the linear combination of the shares in this batch using the same coefficients as those of the j-th share of $[z(i)]_t$ is the same as the share broadcast by P_j for this batch. If both checks pass, P_v broadcasts $(\texttt{accept}, j)$. Otherwise, P_v broadcasts $(\texttt{reject}, j)$.
3. For each party $P_j \in \mathcal{P}_{\texttt{active}}$, if majority broadcast $(\texttt{reject}, j)$, P_j is identified as a corrupted party.

Protocol 34: AnalyzeSharing

1. All parties invoke Localize.
2. In the case that all parties take P_i as output, there are two cases.
   - If $P_i \in \mathcal{P}_{\texttt{active}}$, all parties invoke Active-Dealer.
   - Otherwise, all parties invoke Corrupted-Dealer.

Summary of Analyze Sharing. The full description of Analyze-Sharing appears in Protocol 34.

Note that each call of Analyze-Sharing allows all parties to identify a new corrupted party or a new pair of disputed parties. In particular, each call of Corrupted-Dealer identifies at least one corrupted party. Therefore, in the whole protocol, Localize and Active-Dealer will be invoked at most n^2 times and Corrupted-Dealer will be invoked at most n times. The overall communication complexity of Analyze-Sharing is $O(\ell n^3\phi + n^4\kappa) = O(C\phi + n^4\kappa)$ bits plus $O(n^3\kappa) \cdot \mathcal{BC}$. Here $\ell = O(\frac{m}{t+1})$ is the size of each batch of shares. Note that for any constant $\epsilon > 0$, by setting $\ell = \frac{\epsilon m}{t+1}$, the overall communication complexity of Analyze-Sharing is only $O(\epsilon\phi)$ bits per gate.

6 Protocol

6.1 Handling Input Gates

In this part, we handle the input gates of the circuit. Recall that $m = C/n^2$ is the size of each segment. Each time, at most m input gates are handled. We further require that all inputs in each segment belong to the same party. The description of Input appears in Protocol 35. The communication complexity of input is $O(mn\phi)$ bits.

6.2 Handling Output Gates

We assume that all parties are supposed to receive the same outputs. Each time, at most m output gates are handled. The description of Output appears in Protocol 36. The communication complexity of Output(m) is $O(mn\phi)$ bits plus $O(n\kappa) \cdot \mathcal{BC}$ (not counting Analyze-Sharing).

Lemma 8. *If all parties accept the output, with overwhelming probability, all parties receive the same correct results.*

Protocol 35: INPUT(P_i)

1. Let $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ denote the inputs of P_i.
2. For each $j \in [m]$, P_i samples a random degree-t sharing $[x^{(j)}]_t$ such that the shares held by parties in $\mathcal{D}isp_i$ are set to be 0. Then P_i distributes $[x^{(j)}]_t$ to all other parties.

Protocol 36: OUTPUT(m)

1. A special party $P_{\texttt{king}}$ is selected.
2. Suppose $\{[x^{(i)}]_t\}_{i=1}^m$ are sharings we need to reconstruct. All parties send their shares of $\{[x^{(i)}]_t\}_{i=1}^m$ to $P_{\texttt{king}}$ (via relays if necessary).
3. For each $i \in [m]$, $P_{\texttt{king}}$ checks whether $[x^{(i)}]_t$ is consistent.
   - If $[x^{(i)}]_t$ is inconsistent, $P_{\texttt{king}}$ broadcasts $(\texttt{inconsistent}, i)$. All parties broadcast their shares of $[x^{(i)}]_t$.
     - If $[x^{(i)}]_t$ broadcast by all parties is inconsistent, all parties invoke ANALYZE-SHARING to identify a new corrupted party or a new pair of disputed parties.
     - Otherwise, $P_{\texttt{king}}$ broadcasts $(\texttt{accuse}, j)$ to indicate that P_j didn't send the correct share to $P_{\texttt{king}}$. If $(P_{\texttt{king}}, P_j) \notin \mathcal{D}isp$, $(P_{\texttt{king}}, P_j)$ is a new pair of disputed parties. Otherwise, the relay $P_{\texttt{king}\leftrightarrow j}$ broadcasts its opinion. If $P_{\texttt{king}\leftrightarrow j}$ agrees with $P_{\texttt{king}}$, $(P_{\texttt{king}\leftrightarrow j}, P_j)$ is a new pair of disputed parties. Otherwise $(P_{\texttt{king}\leftrightarrow j}, P_{\texttt{king}})$ is a new pair of disputed parties.
   - Otherwise, $P_{\texttt{king}}$ reconstructs the result $x^{(i)}$ and sends it back to all other parties.
4. **Verify the Reconstructions**
   (a) All parties invoke CHALLENGE to generate $\lambda \in \mathbb{K}$.
   (b) All parties locally compute $[\sigma]_t = \sum_{i=1}^m [x^{(i)}]_t \cdot \lambda^i$ and $\sigma = \sum_{i=1}^m x^{(i)} \cdot \lambda^i$.
   (c) All parties broadcast their shares of $[\sigma]_t$ and σ. Then check the following.
   - If $[\sigma]_t$ is inconsistent, all parties invoke ANALYZE-SHARING to identify a new corrupted party or a new pair of disputed parties.
   - If some party P_i broadcasts a different σ from that broadcast by $P_{\texttt{king}}$, if $(P_i, P_{\texttt{king}}) \notin \mathcal{D}isp$, $(P_i, P_{\texttt{king}})$ is a new pair of disputed parties. Otherwise, if the relay $P_{\texttt{king}\leftrightarrow i}$ broadcasts the same value as $P_{\texttt{king}}$, $(P_{\texttt{king}\leftrightarrow i}, P_i)$ is a new pair of disputed parties. Otherwise, $(P_{\texttt{king}\leftrightarrow i}, P_{\texttt{king}})$ is a new pair of disputed parties.
   - If the secret of $[\sigma]_t$ does not match σ, $P_{\texttt{king}}$ is regarded as a corrupted party.

   If none of above cases happens, all parties accept the output.

6.3 Main Protocol

Now we are ready to present our main construction.

The whole circuit is divided into $O(n^2)$ segments. Recall that C is the size of the circuit and the size of each segment is set to be $m = C/n^2$. There are 4 types of segments:

1. Input-Seg: only contains input gates and all inputs belong to one party.
2. Rand-Seg: only contains rand gates.
3. Comp-Seg: only contains addition gates and multiplication gates.
4. Output-Seg: only contains output gates.

Each time a new corrupted party or a new pair of disputed parties is identified, all parties restart the evaluation of current segment with updated $\mathcal{D}isp, \mathcal{C}orr$. The communication complexity of the whole protocol is $O(Cn\phi + n^3\kappa^2 + n^6\kappa)$ bits plus $O(n^3\kappa) \cdot \mathcal{BC}$.

Protocol 37: MAIN

The whole circuit is divided into $O(n^2)$ segments. For each segment `seg`, all parties do the following steps. If a new corrupted party or a new pair of disputed parties is identified, all parties restart the evaluation of this segment with updated $\mathcal{D}isp, \mathcal{C}orr$.

1. Depending on the type of `seg`, there are 4 cases.
   - If `seg` is an Input-Seg and the inputs belong to P_i, all parties invoke INPUT(P_i).
   - If `seg` is a Rand-Seg, all parties invoke $\frac{m}{t+1}$ times of RAND and take $[r^{(1)}]_t, \ldots, [r^{(m)}]_t$ as output.
   - If `seg` is a Comp-Seg, all parties invoke EVAL(`seg`).
   - If `seg` is an Output-Seg, all parties invoke OUTPUT.
2. For Input-Seg, Rand-Seg and Comp-Seg, all parties invoke VERIFY-SHARING to verify the consistency of the sharings dealt by each party in this segment.
3. All parties add verifiability to the sharings dealt by each party for Input-Seg, Rand-Seg and Comp-Seg. Recall that the size of each batch is set to be $\ell = O(\frac{m}{t+1})$.
   - For an Input-Seg, only the input holder P_i deals $O(m)$ degree-t sharings. All parties invoke $O(n)$ times of $\mathcal{F}_{\text{Tag}}(P_i)$ to authenticate the sharings dealt by P_i.
   - For a Rand-Seg, each party $P_i \in \mathcal{P}_{\texttt{active}}$ deals $O(\frac{m}{t+1})$ random degree-t sharings when invoking RAND. For each party P_i, all parties invoke $O(1)$ times of $\mathcal{F}_{\text{Tag}}(P_i)$ to authenticate the sharings dealt by P_i.
   - For a Comp-Seg, each party $P_i \in \mathcal{P}_{\texttt{active}}$ deals $O(\frac{m}{t+1})$ random degree-t sharings when invoking DOUBLERAND in COMPUTE(`seg`). These sharings are authenticated in the same way as that for a Rand-Seg. In addition, $P_{\texttt{king}}$ deals $O(m)$ degree-t sharings when invoking PARTIALMULT in COMPUTE(`seg`). These sharings are authenticated in the same way as that for an Input-Seg.

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

Concrete Efficiency of Main. We give a brief analysis of the concrete efficiency of MAIN. Recall that each multiplication is done by two steps, REFRESH and PARTIALMULT. The concrete cost of PARTIALMULT is the same as that in [GS20], i.e., 5.5 elements per party.

For REFRESH, all parties need to prepare a degree-t random sharing $[r]_t$ such that only parties in $\mathcal{T}$ receive the shares. By invoking a modified version of RAND, the average cost per sharing is 1 element. Also, each party in $\mathcal{T}$ needs to send one element to $P_{\texttt{king}}$ and $P_{\texttt{king}}$ needs to distribute a degree-t sharing to $\mathcal{T}$. Therefore, the cost of REFRESH is 2 elements per party.

We also need to count the cost of $\mathcal{F}_{\text{Tag}}$ and ANALYZE-SHARING. Recall that $\ell = O(\frac{m}{t+1})$ is the size of each batch of shares we can authenticate each time. For any constant $\epsilon > 0$, by choosing $\ell = \frac{\epsilon m}{t+1}$, the cost of $\mathcal{F}_{\text{Tag}}$ and ANALYZE-SHARING can be reduced to $O(\epsilon)$ elements per gate per party.

In summary, for any fixed constant $\epsilon > 0$, the cost per multiplication gate of MAIN is $(7.5 + \epsilon)$ elements per party. When $\mathcal{C}orr = \emptyset$, we do not need to run REFRESH. Therefore, the cost is reduced to $(5.5 + \epsilon)$ elements per party.

6.4 Construction of the Simulator

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

Suppose $\mathcal{A}$ is the adversary in the real world. In the beginning, we set $\mathcal{P}_{\texttt{active}}$ to be the set of all parties. Let $\mathcal{C}$ denote the set of corrupted parties. Let $\mathcal{H}$ be the set of all honest parties. We always have $\mathcal{H} \subseteq \mathcal{P}_{\texttt{active}}$.

After partitioning the circuit into $O(n^2)$ segments, $\mathcal{S}$ does the following steps for each segment `seg`.

Simulating Step 1. Depending on the type of `seg`, there are 4 cases.

Input-Seg. If `seg` is an Input-Seg and the inputs belong to P_i, $\mathcal{S}$ simulates the behaviors of honest parties in $\textsc{Input}(P_i)$. $\mathcal{S}$ reconstructs the inputs of corrupted parties using the first $t+1$ shares of honest parties.

If P_i is a corrupted party, $\mathcal{S}$ receives the shares of honest parties distributed by P_i.

If P_i is an honest party, for each input $x^{(j)}$, $\mathcal{S}$ sets the shares of parties in $\mathcal{D}isp_i$ to be 0 and the shares of other corrupted parties to be uniformly random elements. Then $\mathcal{S}$ distributes $[x^{(j)}]_t$ to parties in $\mathcal{C}\backslash\mathcal{D}isp_i$.

Rand-Seg. If `seg` is a Rand-Seg, $\mathcal{S}$ simulates the behaviors of honest parties in $\textsc{Rand}$. Whenever an honest party P_i needs to generate a random sharing $[s]_t$, $\mathcal{S}$ sets the shares of parties in $\mathcal{D}isp_i$ to be 0 and the shares of other corrupted parties to be uniformly random elements. Then $\mathcal{S}$ distributes $[s]_t$ to parties in $\mathcal{C}\backslash\mathcal{D}isp_i$.

Comp-Seg. If `seg` is a Comp-Seg, $\mathcal{S}$ simulates the behaviors of honest parties in $\textsc{Eval}(\texttt{seg})$.

Simulating Compute: $\mathcal{S}$ first simulates the behaviors of honest parties in $\textsc{Compute}(\texttt{seg})$ as following.

1. $\mathcal{S}$ behaves honestly when choosing $P_{\texttt{king}}$ and $\mathcal{T}$.
2. If $\mathcal{C}orr = \emptyset$, $\mathcal{S}$ does nothing. Otherwise, for each call of $\textsc{Rand}$, whenever an honest party P_i needs to generate a random sharing $[s]_t$, $\mathcal{S}$ sets the shares of parties in $\mathcal{D}isp_i$ to be 0 and the shares of other corrupted parties to be uniformly random elements. Then $\mathcal{S}$ distributes $[s]_t$ to parties in $\mathcal{T}\backslash\mathcal{D}isp_i$.
3. For each call of $\textsc{DoubleRand}$, whenever an honest party P_i needs to generate a pair of random double sharings $([s]_t, [s]_{2t})$, $\mathcal{S}$ sets the shares of parties in $\mathcal{D}isp_i$ to be 0 and the shares of other corrupted parties to be uniformly random elements. Then $\mathcal{S}$ distributes $([s]_t, [s]_{2t})$ to parties in $\mathcal{C}\backslash\mathcal{D}isp_i$.
4. $\mathcal{S}$ does nothing when computing addition gates.
5. For each call of $\textsc{Mult}([x]_t, [y]_t)$, $\mathcal{S}$ does the following.
   (a) For $\textsc{Refresh}$, $\mathcal{S}$ needs to prepare the shares of $[e]_t := [x]_t + [r]_t$ held by honest parties in $\mathcal{T}$. Recall that $[x]_t$ and $[r]_t$ can be decomposed into $[x]_t = \sum_{i=1}^{n}[x(i)]_t$ and $[r]_t := \sum_{i=1}^{n}[r(i)]_t$, where $[x(i)]_t, [r(i)]_t$ are linear combinations of the sharings dealt by P_i. Let $[x(\mathcal{H})]_t = \sum_{i\in\mathcal{H}}[x(i)]_t$ and $[x(\mathcal{C})]_t = \sum_{i\in\mathcal{C}}[x(i)]_t$. Similarly we can define $[r(\mathcal{H})]_t$ and $[r(\mathcal{C})]_t$. For $[e(\mathcal{C})]_t := [x(\mathcal{C})]_t + [r(\mathcal{C})]_t$, the shares held by honest parties are distributed by corrupted parties and therefore are known to $\mathcal{S}$. For $[e(\mathcal{H})]_t := [x(\mathcal{H})]_t + [r(\mathcal{H})]_t$, the shares held by corrupted parties are distributed by $\mathcal{S}$ and therefore are known to $\mathcal{S}$. $\mathcal{S}$ samples a random sharing as $[e(\mathcal{H})]_t$ based on the shares held by corrupted parties. Then $\mathcal{S}$ computes the shares of $[e]_t = [e(\mathcal{H})]_t + [e(\mathcal{C})]_t$ held by honest parties.
   (b) For $\textsc{PartialMult}$, $\mathcal{S}$ sets the shares of $[e]_{2t}$ held by honest parties to be uniformly random elements.
   (c) If $P_{\texttt{king}}$ is an honest party, $\mathcal{S}$ behaves honestly on behalf of $P_{\texttt{king}}$.

Simulating Check-Refresh and Check-Rand: Then, $\mathcal{S}$ simulates the behaviors of honest parties in $\textsc{Check-Refresh}$ and $\textsc{Check-Rand}$. For $\textsc{Check-Refresh}$, $\mathcal{S}$ does the following.

1. When preparing a random transcript of $\textsc{Refresh}$, $\mathcal{S}$ simulates $\textsc{Rand}$ and $\textsc{Refresh}$ as described above.
2. $\mathcal{S}$ behaves honestly when generating a challenge.
3. $\mathcal{S}$ needs to prepare the shares of $[x]_t, [\tilde{x}]_t, [\tilde{r}]_t, [e]_t, [o]_t$ held by honest parties.
   - Recall that $[x]_t$ can be decomposed into $[x]_t = [x(\mathcal{H})]_t + [x(\mathcal{C})]_t$. The shares of $[x(\mathcal{C})]_t$ held by honest parties are distributed by corrupted parties and therefore are known to $\mathcal{S}$. For $[x(\mathcal{H})]_t$, $\mathcal{S}$ samples a random sharing as $[x(\mathcal{H})]_t$.
   - Recall that $[e]_t = [x]_t + [\tilde{r}]_t$. The shares of $[e]_t$ held by honest parties in $\mathcal{T}$ have been determined since they were explicitly generated and sent to $P_{\texttt{king}}$ when simulating $\textsc{Refresh}$. Therefore, $\mathcal{S}$ is able to compute the shares of $[e]_t, [\tilde{r}]_t$ held by honest parties.
   - The shares of $[o]_t$ held by honest parties are known since they are distributed by $P_{\texttt{king}}$. Note that even if $P_{\texttt{king}}$ is an honest party, $\mathcal{S}$ faithfully simulates $P_{\texttt{king}}$. The shares of $[\tilde{x}]_t$ held by honest parties can be computed by $[\tilde{x}]_t = [x]_t - [o]_t$.
4. $\mathcal{S}$ honestly broadcasts the shares held by honest parties.
5. $\mathcal{S}$ honestly checks the sharings. The simulation for $\textsc{Analyze-Sharing}$ and $\textsc{Check-Rand}$ will be introduced later.

For Check-Rand, $\mathcal{S}$ first prepares the shares of $[\tilde{r}(i)]_t$ held by honest parties for each i. For a corrupted party P_i, the shares of $[\tilde{r}(i)]_t$ held by honest parties are distributed by P_i and therefore are known to $\mathcal{S}$. $\mathcal{S}$ computes the shares of $[\tilde{r}(\mathcal{H})]_t$ held by honest parties by $[\tilde{r}(\mathcal{H})]_t = [\tilde{r}]_t - \sum_{i\in\mathcal{C}}[\tilde{r}(i)]_t$. Suppose $P_{i^\star} \in \mathcal{H}$ is the first honest party. For each honest party $P_i \in \mathcal{H}\backslash\{P_{i^\star}\}$, $\mathcal{S}$ samples a random sharing as $[\tilde{r}(i)]_t$ based on the shares held by corrupted parties. Then the sharing $[\tilde{r}(i^\star)]_t$ is set to be $[\tilde{r}(\mathcal{H})]_t - \sum_{i\in\mathcal{H}\backslash\{P_{i^\star}\}}[\tilde{r}(i)]_t$. $\mathcal{S}$ then faithfully follows the protocol Check-Rand.

Simulating the Verification for PartialMult: Finally, $\mathcal{S}$ simulates the behaviors of honest parties when verifying PartialMult. Note that Extend-PartialMult can be simulated in a similar way to PartialMult. Since the communication are only required when invoking Extend-PartialMult and Challenge in Compress, $\mathcal{S}$ simulates these two protocols as described above when simulating Compress.

- For De-Linearization, the communication is only required when invoking Challenge. $\mathcal{S}$ behaves honestly when invoking Challenge.
- For Dimension-Reduction, it can be simulated by simulating Extend-PartialMult and Compress as described above.
- For Randomization, $\mathcal{S}$ simulates Rand, DoubleRand and PartialMult when preparing a random transcript of PartialMult. The rest of steps can be simulated in the same way as that in Dimension-Reduction.

For Check-Single-Mult, $\mathcal{S}$ needs to prepare the shares of $[\alpha]_t, [\beta]_t, [\delta]_t, [\delta]_{2t}, [\eta]_{2t}, [\eta]_t, [\gamma]_t$ held by honest parties.

- For $[\alpha]_t, [\beta]_t$, the shares held by honest parties can be prepared as that in Check-Refresh.
- Recall that $[\eta]_{2t} = [\alpha]_t \cdot [\beta]_t + [\delta]_{2t}$ and the shares of $[\eta]_{2t}$ held by honest parties have been determined since they were explicitly generated and sent to $P_{\texttt{king}}$ when simulating PartialMult and Extend-PartialMult. $\mathcal{S}$ computes the shares of $[\eta]_{2t}, [\delta]_{2t}$ held by honest parties.
- Recall that $[\delta]_{2t}$ can be decomposed into $[\delta]_{2t} = [\delta(\mathcal{H})]_{2t} + [\delta(\mathcal{C})]_{2t}$. Since the shares of $[\delta(\mathcal{C})]_{2t}$ held by honest parties are distributed by corrupted parties and therefore are known to $\mathcal{S}$, $\mathcal{S}$ computes the shares of $[\delta(\mathcal{H})]_{2t}$ held by honest parties. Note that the shares of $[\delta(\mathcal{H})]_{2t}$ held by corrupted parties are explicitly generated and distributed by $\mathcal{S}$. Therefore, $\mathcal{S}$ reconstructs the whole sharing $[\delta(\mathcal{H})]_{2t}$ and learns the secret value $\delta(\mathcal{H})$. As for $[\delta(\mathcal{H})]_t$, $\mathcal{S}$ samples a random sharing based on the secret value $\delta(\mathcal{H})$ and the shares held by corrupted parties. Finally, $[\delta]_t$ can be computed by $[\delta]_t = [\delta(\mathcal{H})]_t + [\delta(\mathcal{C})]_t$.
- Recall that $[\gamma]_t = [\eta]_t - [\delta]_t$. The shares of $[\eta]_t$ held by honest parties are explicitly distributed by $P_{\texttt{king}}$ and therefore are known to $\mathcal{S}$. $\mathcal{S}$ computes the shares of $[\eta]_t, [\gamma]_t$ held by honest parties.

Then, $\mathcal{S}$ honestly broadcasts the shares held by honest parties and checks the sharings. The simulation for Analyze-Sharing and Check-DoubleRand will be introduced later.

For Check-DoubleRand, it can be simulated in a similar way to Check-Rand.

Output-Seg. If `seg` is an Output-Seg, $\mathcal{S}$ simulates the behaviors of honest parties in Output(m). $\mathcal{S}$ first sends the inputs of corrupted parties to the functionality of Main and receives the outputs. Then $\mathcal{S}$ simulates Output(m) as following.

1. $\mathcal{S}$ behaves honestly when selecting $P_{\texttt{king}}$.
2. For each output sharing $[x]_t$, $\mathcal{S}$ prepares the shares held by honest parties. Recall that $[x]_t$ can be decomposed into $[x]_t = [x(\mathcal{H})]_t + [x(\mathcal{C})]_t$. $\mathcal{S}$ reconstructs the value $x(\mathcal{C})$ by using the first $t+1$ shares of honest parties. Then $\mathcal{S}$ samples a random sharing as $[x(\mathcal{H})]_t$ based on the secret value $x(\mathcal{H}) = x - x(\mathcal{C})$ and the shares held by corrupted parties. $\mathcal{S}$ computes the shares of $[x]_t$ held by honest parties by $[x]_t = [x(\mathcal{H})]_t + [x(\mathcal{C})]_t$ and sends the shares to $P_{\texttt{king}}$.
3. If $P_{\texttt{king}}$ is an honest party, $\mathcal{S}$ faithfully follows the protocol.
4. Since $\mathcal{S}$ has already generated all the shares of honest parties, $\mathcal{S}$ faithfully follows the rest of steps. The simulation for Analyze-Sharing will be introduced later.

Simulating Step 2. $\mathcal{S}$ simulates VERIFY-SHARING as following.

1. $\mathcal{S}$ simulates RAND as described above to prepare a random degree-t sharing $[s^{(0)}]_t$.
2. $\mathcal{S}$ behaves honestly when generating the challenge.
3. $\mathcal{S}$ computes the shares of $[\sigma]_t$ held by honest parties. Recall that $[\sigma]_t$ can be decomposed into $[\sigma]_t = [\sigma(\mathcal{H})]_t + [\sigma(\mathcal{C})]_t$. $\mathcal{S}$ samples a random sharing as $[\sigma(\mathcal{H})]_t$ based on the shares held by corrupted parties. Since the shares of $[\sigma(\mathcal{C})]_t$ held by honest parties are distributed by corrupted parties, $\mathcal{S}$ can compute the shares of $[\sigma]_t$ held by honest parties.
4. $\mathcal{S}$ honestly follows the protocol. The simulation for ANALYZE-SHARING will be introduced later.

Simulating Step 3. For each call of $\mathcal{F}_{\text{Tag}}$, $\mathcal{S}$ simulates the functionality $\mathcal{F}_{\text{Tag}}$ as following.

1. $\mathcal{S}$ behaves honestly if receiving $(\texttt{disputed}, P_i, P_j)$ and $(\texttt{corrupted}, P_i)$.
2. If P_d is a corrupted party, $\mathcal{S}$ receives the whole sharings $\{[s^{(j)}(d)]_t\}_{j=1}^{\ell}$ from P_d. If P_d is an honest party, $\mathcal{S}$ provides the shares of $\{[s^{(j)}(d)]_t\}_{j=1}^{\ell}$ held by corrupted parties $\mathcal{C}$ on behalf of P_d.
3. For each pair of parties (P_v, P_i) such that $(P_v, P_i), (P_i, P_d) \notin \mathcal{D}isp$, $\mathcal{S}$ does the following.
   (a) If P_v is corrupted, $\mathcal{S}$ receives $(\boldsymbol{\mu}_{v\to i}, \nu)$ from P_v.
   (b) Depending on whether P_v, P_i are corrupted, there are 4 cases:
   - If P_v, P_i are honest, $\mathcal{S}$ does nothing.
   - If P_v is honest but P_i is corrupted, $\mathcal{S}$ samples a random field element as the tag and sends it to P_i. $\mathcal{S}$ records $(\tau, \boldsymbol{\sigma}_i(d))$.
   - If P_i is honest but P_v is corrupted, $\mathcal{S}$ keeps $(\boldsymbol{\mu}_{v\to i}, \nu)$.
   - If P_v, P_i are corrupted, $\mathcal{S}$ faithfully computes τ and sends τ to P_i.

   (c) If P_v, P_i are honest, $\mathcal{S}$ samples random field elements as $(\boldsymbol{\mu}_{v\to i}, \nu)$ and sends the keys to the adversary.

Simulating Analyze-Sharing. Now we are ready to present the simulation for ANALYZE-SHARING.

Simulating LOCALIZE. $\mathcal{S}$ first simulates LOCALIZE.

1. $\mathcal{S}$ simulates RAND as described above when preparing a random degree-t sharing $[r]_t$. Recall that $[r]_t$ can be decomposed into $[r]_t = [r(\mathcal{H})]_t + [r(\mathcal{C})]_t$. $\mathcal{S}$ samples a random sharing as $[r(\mathcal{H})]_t$ and computes the shares of $[r]_t$ held by honest parties.
2. $\mathcal{S}$ honestly follows the protocol. Note that the whole sharing $[z]_t$ has been broadcast. $\mathcal{S}$ prepares the shares of $[z(i)]_t$ held by honest parties for each $i \in [n]$. For a corrupted party P_i, the shares of $[z(i)]_t$ held by honest parties are distributed by P_i and therefore are known to $\mathcal{S}$. Suppose $P_{i^\star} \in \mathcal{H}$ is the first honest party. For each honest party $P_i \in \mathcal{H}\backslash\{P_{i^\star}\}$, $\mathcal{S}$ samples a random sharing as $[z(i)]_t$ based on the shares held by corrupted parties. Then the sharing $[z(i^\star)]_t$ is set to be $[z]_t - \sum_{i\in[n]\backslash\{i^\star\}}[z(i)]_t$.
3. $\mathcal{S}$ faithfully follows the rest of steps in LOCALIZE.

Simulating ACTIVE-DEALER. In the case that $P_i \notin \mathcal{C}orr$, $\mathcal{S}$ simulates the behaviors of honest parties in ACTIVE-DEALER. Note that in the first step, we will locate a pair of disputed parties (P_i, P_j) (which may have already been identified). If P_j is honest but P_i is corrupted, then $\mathcal{S}$ learns all shares held by P_j distributed by P_i. If P_i is honest but P_j is corrupted, then $\mathcal{S}$ has explicitly generated and distributed the shares to P_j. Therefore, in either case, $\mathcal{S}$ is able to honestly follow the protocols in the first two steps to locate a batch of shares which P_i, P_j do not agree on.

In Step 3, if P_v, P_j are honest, $\mathcal{S}$ broadcasts `accept` on behalf of P_v. If P_v is honest but P_j is corrupted, $\mathcal{S}$ receives the batch of shares and the tag from P_j. $\mathcal{S}$ then checks whether the tag matches the one $\mathcal{S}$ recorded when simulating $\mathcal{F}_{\text{Tag}}$ and broadcast `accept` or `reject` faithfully. If P_j is honest but P_v is corrupted, $\mathcal{S}$ uses the authentication keys $(\boldsymbol{\mu}_{v\to j}, \nu)$ stored when simulating $\mathcal{F}_{\text{Tag}}$ to compute the tag of the shares P_j hold. Then $\mathcal{S}$ sends the batch of shares and the tag to P_v. The rest of steps are followed honestly.

Simulating Corrupted-Dealer. In the case that $P_i \in \mathcal{C}orr$, $\mathcal{S}$ has learned all shares of honest parties distributed by P_i. $\mathcal{S}$ faithfully follows the first step to locate a batch of sharings such that at least one of them is inconsistent. Step 2 can be simulated in the same way as Step 3 in Active-Dealer. The rest of steps are followed honestly.

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

6.5 Proof of the Security

In this part, we prove Theorem 1. Formally,

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

Proof. We show that, the view of the adversary $\mathcal{A}$ when interacting with the simulator $\mathcal{S}$ we constructed in Section 6.4 has the same distribution as that in the real world with all but a negligible probability. Consider the following hybrids.

Hybrid$_0$: Execution in the real world.

Hybrid$_1$: In this hybrid, $\mathcal{S}$ honestly follows $\mathcal{F}_{\text{Tag}}$ and records the tags sent to corrupted parties. Then Analyze-Sharing is simulated by $\mathcal{S}$. The two hybrids differ only in the case during the process of Analyze-Sharing when a corrupted party provides incorrect shares with a valid authentication tag to an honest party. In this case, $\mathcal{S}$ would reject this verification while an honest party would accept it in the real world. However, since the authentication keys of honest parties are uniformly distributed, this happens with negligible probability. Therefore, **Hybrid$_0$** is statistically close to **Hybrid$_1$**.

Hybrid$_2$: In this hybrid, $\mathcal{F}_{\text{Tag}}$ is simulated by $\mathcal{S}$. The distribution is the same as **Hybrid$_1$**.

Hybrid$_3$: In this hybrid, Verify-Sharing is replaced by the simulation of $\mathcal{S}$. Note that if the sharings are inconsistent, with overwhelming probability, Verify-Sharing can detect it. The distribution is statistically close to **Hybrid$_2$**. (Although $\mathcal{S}$ perfectly simulates Verify-Sharing, if Verify-Sharing fails to detect inconsistency sharings, other protocols will not be able to simulated by $\mathcal{S}$. Therefore, we assume a failure detection is a failure simulation, which happens with negligible probability.)

Hybrid$_4$: In this hybrid, Output is replaced by the simulation of $\mathcal{S}$. Note that, $\mathcal{S}$ can compute the inputs of corrupted parties when executing Input. According to Lemma 8, the distribution is statistically close to **Hybrid$_3$**.

Hybrid$_5$: In this hybrid, De-Linearization, Dimension-Reduction, Randomization and Check-Single-Mult (which are invoked in Eval) are simulated by $\mathcal{S}$. The distribution is statistically close to **Hybrid$_4$**. The negligible probability comes from the case when the challenge inner-product tuple (or multiplication tuple) is correct while one of the original multiplication tuples is incorrect.

Hybrid$_6$: In this hybrid, Check-Refresh (which is invoked in Eval) is simulated by $\mathcal{S}$. The distribution is statistically close to **Hybrid$_5$**. The negligible probability comes from the case when the challenge transcript is correct while one of the original transcripts is incorrect.

Hybrid$_7$: In this hybrid, Compute (which is invoked in Eval) is simulated by $\mathcal{S}$. The distribution is identical to **Hybrid$_6$**.

Hybrid$_8$: In this hybrid, Input is simulated by $\mathcal{S}$. The distribution is the same as **Hybrid$_7$**.

Hybrid$_9$: In this hybrid, Rand and DoubleRand are simulated by $\mathcal{S}$. The distribution is identical to **Hybrid$_8$**.

Note that **Hybrid$_9$** is the execution between $\mathcal{S}$ and $\mathcal{A}$ in the ideal world.

We conclude that the distribution of **Hybrid$_9$** is statistically close to **Hybrid$_0$**.

References

ABF+17. Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority mpc for malicious adversariesbreaking

the 1 billion-gate per second barrier. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 843–862. IEEE, 2017.

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

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.

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.

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

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

GS20. Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority mpc. Cryptology ePrint Archive, Report 2020/134, 2020. https://eprint.iacr.org/2020/134.

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.