Malicious Security Comes Free in SPDZ

Junru Li
Tsinghua University
jr-li24@mails.tsinghua.edu.cn

Yifan Song
Tsinghua University
and Shanghai Qi Zhi Institute
yfsong@mail.tsinghua.edu.cn

Abstract

We study the communication complexity of SPDZ-type MPC protocols in the dishonest majority setting with maximum corruption, where $t = n - 1$ out of n parties can be corrupted. In the semi-honest setting, the state-of-the-art SPDZ protocol achieves an amortized communication cost of $4n$ field elements per multiplication gate, assuming a pseudorandom correlation generator (PCG) that prepares random Beaver triples silently in the offline phase. However, achieving security against malicious adversaries typically incurs a substantial overhead. Existing works either blow up the communication by incurring $10n$ field elements per gate or an additional overhead of $O(\sqrt{|C|} \cdot n^3)$, or additionally require an additively homomorphic encryption scheme.

In this work, we construct a maliciously secure SPDZ-type protocol with the same amortized communication complexity as the semi-honest variant, i.e., $4n$ field elements per multiplication gate, assuming a PCG for *tensor-product* correlations. We note that the tensor-product correlations come free in communication as by-products from almost all existing constructions of PCGs for random Beaver triples, although additional local computation is required. These by-products allow us to construct an efficient preprocessing phase and a recursive check protocol, which enables an online evaluation of the circuit without authenticating all the wire values. The recursive check requires sublinear communication in $|C|$, though it incurs an $O(k)$ overhead on the online computational cost compared to the semi-honest baseline.

Contents

1 Introduction **3**
- 1.1 Our Contributions . . . 4

2 Preliminaries **6**

3 Technical Overview **7**
- 3.1 Review of SPDZ Protocol . . . 7
- 3.2 Generating Partially Authenticated Beaver Triples without Linear Errors . . . 8
- 3.3 Online Evaluation with Partially Authenticated Beaver Triples . . . 10
- 3.4 Protocol Summary and Further Optimization . . . 11

4 Generating Partially Authenticated Tensor Products **13**
- 4.1 Functionality for Partially Authenticated Tensor Products . . . 13
- 4.2 Construction from Unauthenticated Tensor Products . . . 13
- 4.3 Instantiation of Unauthenticated Tensor Products . . . 15
- 4.4 PCG Instantiations . . . 16
- 4.5 Preprocessing Phase . . . 18

5 Evaluation Phase **19**

6 Verification Phase **20**

7 Main Protocol **23**

8 Further Reducing Communication Cost via Turbospeedz **23**

A Standard Subprotocols **29**

B MAC Check on Opened Values **30**

C Security Proof for the Preparation of Partially Authenticated Tensor Products **30**

D Security Proof for the Preparation of Unauthenticated Tensor Products **36**
- D.1 Instantiation from 2-party tensor product correlations . . . 36
- D.2 Instantiation from n-party unauthenticated triples . . . 37

E Regular Dual-LPN Assumption **39**

F Preprocessing Functionality **40**

G Preprocessing Protocol and Security Proof **40**

H The Protocol Description of Π_{main} **44**

I Cost Analysis **54**

J The Functionality of the Preprocessing for Π_{turbo} **55**

K The Verification Phase of Π_{turbo} **56**

L Protocol Description of Π_{turbo} **56**

M Proof Sketch for the Security of Π_{turbo} **57**

1 Introduction

Secure multiparty computation (MPC) [Yao82, GMW87, CCD88, BGW88, RB89] enables a set of n mutually distrustful parties to jointly evaluate a function on their private inputs, while revealing nothing beyond the prescribed output. Over the decades, the efficiency of MPC protocols has been greatly improved, turning them from purely theoretical primitives into practical cryptographic tools.

The security of an MPC protocol is typically characterized by two primary dimensions: the corruption threshold and the adversarial model. Protocols with an honest majority (corruption threshold $t < n/2$) can achieve information-theoretic security. A long line of works [BGW88, DN07, LN17, CGH+18, GS20, BGIN20, GLO+21, EGPS22] has focused on optimizing the communication efficiency in this setting. Notably, the state-of-the-art amortized communication cost is $4n$ field elements per multiplication gate, applicable to both semi-honest security and malicious security with abort [GLO+21].

In contrast, MPC protocols allowing for a dishonest majority (up to $t = n - 1$) address the worst-case corruption scenario. It is well-established that achieving unconditional security is impossible in this setting [BGW88]. To avoid using heavyweight public-key operations, practical solutions for dishonest majority MPC often consider the preprocessing model. In this paradigm, parties generate correlated randomness in an offline phase, enabling a highly efficient, information-theoretic online phase. Specifically, the seminal SPDZ protocol [DPSZ12] introduced an online phase with communication complexity linear in the number of parties, assuming preprocessed random Beaver triples in the offline phase. To be more concrete, the online communication is $4|C|n$ field elements for computing a circuit of size $|C|$. Subsequently, numerous works [DKL+13, LOS14, KOS16, KPR18, BCS19, BNO19, BGIN21] have significantly improved upon the SPDZ paradigm in both the preprocessing and online phases. Leveraging recent advancements in pseudorandom correlation generators (PCGs) [BCGI18, BCG+19, BCG+20, WYKW21], an amortized cost of $O(n)$ field elements per multiplication gate can be achieved across both phases even for malicious security [BGIN22, RS22, AS22, LXYY26].

A semi-honest secure SPDZ protocol can achieve an amortized communication cost of $4n$ field elements per multiplication gate simply from PCGs for Beaver triples. However, achieving malicious security typically incurs a large communication or computation overhead. In particular, the best-known malicious SPDZ protocols are constructed in two different ways:

- One direction is to prepare authenticated Beaver triples [RS22, AS22, GLM+24, LXYY26] during a circuit-independent preprocessing phase. However, current constructions of PCGs do not support the correlation of fully authenticated triples, so additional communication is required for the authentication steps. In particular, the work [AS22] achieves communication of $4n$ field elements per gate [GLM+24] that matches the semi-honest baseline. However, it requires PCG for directly preparing authenticated triples, which incurs a large additional communication overhead of $O(\sqrt{|C|} \cdot n^3)$ during the preprocessing. On the other hand, the work [RS22] requires communication of $12n$ field elements per gate, where the additional communication cost is similar to the semi-honest baseline, which is only *quadratic* in n. Recent works [GLM+24, LXYY26] further reduce the cost to $10n$ field elements per gate, but it remains significantly less efficient than in the semi-honest setting.

- The other direction is to evaluate the circuit as the semi-honest SPDZ protocol, while using zero-knowledge on distributed data (D-ZK) to ensure the correctness of the computation [BGIN21, BGIN22, BGH+25]. This approach can achieve $4n$ elements per multiplication gate, matching the semi-honest SPDZ protocol. However, such an approach either requires a *circuit-dependent* preprocessing phase [BGIN21, BGIN22] or operates in the $(n+1)$-party model with an offline dealer that does not collude with online parties [BGIN22]. In the standard n-party dishonest-majority setting, only [BGIN22] instantiates the preprocessing, while additionally assuming an additively homomorphic public-key encryption (AHE) scheme. Compared to the semi-honest SPDZ protocol, $O(|C|)$ homomorphic operations and $O(\sqrt{|C|})$ AHE encryptions are required, resulting in a significant computational overhead.

A natural question then arises:

"Can we lift the semi-honest SPDZ protocol to malicious security with the same amortized communication complexity per multiplication gate without additional assumptions while maintaining the quadratic communication overhead in n?"

1.1 Our Contributions

In this work, we give a positive answer to the above question. In particular, we present a maliciously secure SPDZ-type MPC protocol for arithmetic circuits over large fields ($\log|\mathbb{F}| = \Omega(\kappa)$) in the dishonest majority setting ($t = n-1$) with the following properties:

- **Amortized Communication Cost.** Our construction achieves an amortized communication cost of $4n$ field elements per multiplication gate, matching the cost of the semi-honest variant. Using LPN-based programmable PCGs for tensor product correlations, the additional communication cost remains quadratic in n. This represents a significant improvement over the state-of-the-art results [AS22, RS22, GLM+24, LXYY26] without additively homomorphic encryption schemes or circuit-dependent preprocessing.
- **Silent Preprocessing Generation.** For a parameter k, our construction requires $|C|/k + o(|C|/k)$ groups of preprocessing data in the form of $(\{[a_i], [b_i]\}_{j=1}^{k}, \{[a_i \cdot b_j]\}_{i,j=1}^{k})$[1], where $[\cdot]$ denotes an additive sharing. We refer to this type of preprocessing data as a length-k tensor product. Our preprocessing data is independent of the circuit topology and can be silently generated by almost all existing PCGs for random Beaver triples. The procedure for upgrading the Beaver triples to the tensor products is almost free of communication, but may require additional computation.
- **Information-Theoretic Online Phase.** Our online phase maintains the information-theoretic security and only contains the semi-honest SPDZ online protocol plus an efficient verification on the correctness of evaluation, following the recursive check of [GS20] in the honest majority setting. The verification requires $O(|C|n \cdot \frac{\log k}{k})$ communication and $O(|C|nk)$ computational cost. When $k = w(1)$, the amortized communication complexity per gate remains $4n$ elements and matches the semi-honest baseline, while the computational cost has an $O(k)$ overhead over the semi-honest SPDZ protocol.

We further show that the amortized communication cost can be reduced to $2n$ per multiplication gate by employing the Turbospeedz online protocol [BNO19] in our framework, albeit at the cost of increased computational complexity (quadratic in $|C|$). This may suit the case for small circuits or SIMD circuits.

To achieve this result, we revisit the current construction of malicious SPDZ-type protocols from authenticated Beaver triples [RS22, GLM+24]. Recall that an authenticated Beaver triple is in the form of $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$ where $\llbracket \cdot \rrbracket$ denotes the *authenticated* additive sharing, a, b are random field elements, and $c = a \cdot b$. In [RS22], the authors construct a PCG that can silently prepare *partially* authenticated Beaver triples in the form of $(\llbracket a \rrbracket, \llbracket b \rrbracket, [c])$, i.e., only the first two sharings are authenticated. In addition, their PCG allows an adversary to insert a linear error on c by selecting δ_a, δ_b and causing $c = a \cdot b + \delta_a \cdot a + \delta_b \cdot b$. The main overhead compared to the semi-honest SPDZ protocol comes from (1) the verification of partially authenticated Beaver triples, and (2) the authentication of c sharings.

We observe that when preparing a group of length-k partially authenticated Beaver triples $\{(\llbracket a_i \rrbracket, \llbracket b_i \rrbracket, [c_i])\}_{i=1}^{k}$ using the PCG in [RS22], parties can also *locally* compute shared cross product $[a_i \cdot b_j]$ for all $i, j \in \{1, \ldots, k\}$. We refer to this type of preprocessing data as a length-k partially authenticated tensor product. With these by-products from the PCG in [RS22], we construct an efficient verification that detects linear errors with sublinear communication in k, which solves the first issue. Furthermore, we show that length-k partially authenticated tensor products can be computed from length-$(k+1)$ unauthenticated tensor products in the form of $(\{[a_i]\}_{i=1}^{k+1}, \{[b_i]\}_{j=1}^{k+1}, \{[a_i \cdot b_j]\}_{i,j=1}^{k+1})$ in a black-box way, so we only need to prepare unauthenticated tensor products during the preprocessing.

[1] In our protocol construction with parameter k, we actually prepare $(\{[a_i]\}_{i=1}^{k+1}, \{[b_i]\}_{j=1}^{k+1}, \{[a_i \cdot b_j]\}_{i,j=1}^{k+1})$. We omit the difference in the introduction.

To avoid the overhead of authenticating the c sharings, our idea is to directly evaluate the online phase using partially authenticated Beaver triples. We note that in this way, the inputs of all multiplication gates are authenticated during the computation. Thus, it suffices to verify that all authenticated wire values satisfy the correct correlations defined by the circuit. This, in turn, can be translated to a verification of an inner-product correlation. For this verification, we adapt the recursive check framework of [GS20], originally designed for the honest majority setting, which only incurs a sublinear communication in the circuit size. However, the communication efficiency of the technique in [GS20] crucially relies on the fact that in the honest majority setting, an inner-product operation over shared data can be computed at the same cost as a single multiplication operation, which does not hold in the dishonest majority setting. To overcome this obstacle, we again leverage the length-k tensor products and show that each block of k multiplication gates can be verified with sublinear communication in k. When $k = \omega(1)$, we obtain the desired result.

We refer the readers to Section 3 for a more detailed overview of these techniques.

Preprocessing Cost. We give two different approaches for generating tensor-product correlations during the preprocessing. The first one uses 2-party tensor products, which can be instantiated from several existing PCG constructions for 2-party oblivious linear evaluations (OLEs) via local computation without additional communication. Currently, one of the most efficient ways of generating Beaver triples via PCG is also using 2-party OLEs. Note that an instance of PCG may prepare $m > k$ OLEs each time. In principle, parties get a length-m tensor product via local computation. The seed length is $O(|C|n^2 \cdot \mathsf{poly}(\log m)/m)$ and the computational cost is $O(|C|n^2 m \cdot \mathsf{poly}(\kappa, \log m))$. If only assuming the standard LPN assumption, the computational complexity is comparable to that of the PCGs for Beaver triples [BCG+19], but there will be an $O(m)$ computational overhead if assuming ring-LPN-type assumptions [BCG+20]. For better online computation efficiency, we may parse it into m/k length-k tensor products. This is the precise sense in which tensor-product correlations come *free in communication*. When taking $m = O(|C|)$ to generate all the $O(|C|/k)$ length-k tensor products, the communication cost is quadratic in n and polylogarithmic in $|C|$, which is significantly more efficient than directly consuming $O(n^3\sqrt{|C|})$ communication to generate fully authenticated triples [AS22].

Our second instantiation is the one following [BCG+20] that treats a programmable PCG for unauthenticated Beaver triples as a black box. This version requires $O(k^2)$ instances of the PCG for triples. It incurs $O(n^2k^2 \cdot \mathsf{poly}(\log |C|))$ additional communication and $O(|C|n^2k \cdot \mathsf{poly}(\kappa, \log |C|))$ computation for typical PCGs for triples from ring-LPN-type assumptions. Nevertheless, the additional communication remains quadratic in n and sublinear in $|C|$, and the construction requires no cryptographic assumption beyond that of the underlying triple PCG. Thus, this generic approach is black-box and *almost free in communication* (though not in computation either).

Concurrent Work. We note that a concurrent work [BGH+26] also achieves a similar $4n$ per-gate amortized communication cost in a maliciously secure SPDZ protocol with a sublinear communication overhead that is quadratic in n. The main difference between the two works concerns generality versus computation under specific assumptions. [BGH+26] prepares fully authenticated Beaver triples efficiently, which strongly relies on specific PCGs based on LPN-style assumptions in the preprocessing phase. In comparison, what we provide is a more generalized approach, which instead identifies tensor-product correlations as the common resource. It therefore supports instantiations from broader assumptions. To achieve the same asymptotic communication overhead in the circuit size $|C|$, our protocol requires a slightly higher computational cost than that of [BGH+26] if assuming Ring-LPN [BCG+20] or Search-QA-SD [BCCD23] assumptions. The communication and computational cost of the two works are similar if only assuming standard LPN-type assumptions.

We provide a comparison between the semi-honest baseline, our protocols, the malicious SPDZ-type protocols from [RS22, GLM+24, LXYY26, AS22, BGIN22], and the concurrent work [BGH+26] in Table 1.

Protocol	Circuit Dependency	Preprocessing from PCG	Comm. per Gate	Additional Assumption Beyond PCG
Semi-honest baseline	No	$\|C\|$ triples	$4n$	/
[RS22] + [GLM+24] or [LXYY26]	No	$2\|C\|$ PA triples	$10n$	/
[AS22] or [BGH+26]	No	$\|C\|$ AT triples	$4n$	/
[BGIN22]	Yes	$\|C\|$ triples	$4n$	AHE
Ours	No	$\|C\|/k$ length-k tensor products	$4n + O(n \cdot \frac{\log k}{k})$	/
Ours	No	a length-$\|C\|$ tensor product	$2n$	/

Table 1: Comparison of malicious SPDZ-type protocols. PA stands for partially authenticated, and AT stands for authenticated. A partially authenticated Beaver triple is in the form of $([\![a]\!], [\![b]\!], [c])$, i.e., only the first two additive sharings are authenticated. [AS22] requires authenticated Beaver triple in the form of $([\![a]\!], [\![b]\!], [\![c]\!])$, while [BGH+26] requires unauthenticated multiplication triples generated by a sparse-tensored PCG, together with authenticated shares of the nonzero positions and payloads of the underlying sparse vectors, which can be regarded as fully authenticated using the verification process via PIR in [BGH+26]. AHE stands for additively homomorphic public-key encryption. We omit sublinear terms on the number of preprocessing data from PCG. The last column shows the additional assumption beyond the PCG required in the protocol.

2 Preliminaries

Notation. Let κ denote the security parameter. $\mathbb{F}$ is a finite field with $\log |\mathbb{F}| = \Omega(\kappa)$. We consider MPC protocols run among n parties for an arithmetic circuit C over $\mathbb{F}$. We use $[x]$ to denote the additive sharing for a value x among the parties, and we use $[\![x]\!]$ to denote the authenticated sharing $[\![x]\!] = ([x], [\Delta], [\Delta \cdot x])$ among them. We use $\langle \boldsymbol{u}, \boldsymbol{v} \rangle, \boldsymbol{u} * \boldsymbol{v}, \boldsymbol{u} \otimes \boldsymbol{v}$ to denote the inner product, the coordinate-wise product, and the tensor product of two vectors $\boldsymbol{u}, \boldsymbol{v}$, respectively.

Security Model. We define the security of multiparty computation in the *real and ideal world paradigm* [Can00]. We consider a protocol Π to be secure if any adversary's view in its execution in the real world can also be simulated in the ideal world. We consider malicious security with abort against a static adversary $\mathcal{A}$ corrupting up to $t = n - 1$ parties. We denote the n parties by $P_1, \ldots, P_n$.

Standard Subprotocols. We use standard functionalities for subprotocols $\mathcal{F}_{\mathsf{Commit}}$ of letting a party commit to a message and $\mathcal{F}_{\mathsf{Coin}}$ of generating a random public coin in $\mathbb{F}^*$. The detailed descriptions of the functionalities are given in Section A.

MAC Check on Opened Values. With random coins and commitments, the opened authenticated values are able to be checked. In particular, for an authenticated sharing $[\![x]\!]$ whose secret x needs to be reconstructed to all the parties, the parties can call a protocol $\Pi_{\mathsf{Open}}([\![x]\!])$ to let a party P_{king} collect all parties' shares of $[x]$ and send x to all parties. All the secrets of authenticated sharing opened via Π_{Open}

can be checked together via a protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ [DKL+13]. It is guaranteed that if a secret x of $[\![x]\!]$ is not opened correctly, honest parties will abort the protocol during $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ with an overwhelming probability. For more details about Π_{Open} and $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, we refer the readers to Section B.

3 Technical Overview

We give a high-level overview of the techniques used in this work. We use $[x]$ to denote an additive sharing of x over all n parties. We use $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ to denote the inner product of two vectors $\boldsymbol{x}, \boldsymbol{y}$. We focus on maliciously secure dishonest majority MPC ($t = n - 1$) for an arithmetic circuit C with $|C|$ multiplication gates over a large field $\mathbb{F}$ with $\log |\mathbb{F}| = \Omega(\kappa)$. The communication cost is measured by the number of field elements.

3.1 Review of SPDZ Protocol

We first briefly review the state-of-the-art malicious SPDZ-type protocol from [RS22, GLM+24].

The SPDZ Template. A (semi-honest) SPDZ-type MPC [DPSZ12] operates in two phases. In the *circuit-independent* preprocessing phase, parties together generate $|C|$ random Beaver triples in the form of $([a], [b], [c])$ where a, b are random field elements, and $c = a \cdot b$. From [RS22], random Beaver triples can be generated via a pseudorandom correlation generator (PCG) [BCG+19, BCG+20, WYKW21] with $o(1)$ amortized communication cost per triple.

Then, in the online phase, parties evaluate the circuit gate-by-gate and compute an additive secret sharing for each wire in the circuit. For each addition gate with input sharings $([x], [y])$, parties can locally add up their shares to obtain $[z] := [x] + [y]$. For each multiplication gate, a random Beaver triple $([a], [b], [c])$ is consumed:

1. All parties locally compute $[x + a] := [x] + [a], [y + b] := [y] + [b]$ and send their shares to a single party P_{king}.
2. P_{king} reconstructs the secrets $x + a, y + b$ and sends the result to all parties.
3. Then all parties locally compute the output sharing $[z] := (x+a) \cdot (y+b) - (x+a) \cdot [b] - (y+b) \cdot [a] + [c]$.

As a result, the online phase only involves 2 public reconstructions per multiplication gate, which incurs $4n$ field elements of communication.

Towards Malicious Security. To achieve malicious security, [DKL+13] employs information-theoretic MACs to authenticate additively shared values. Specifically, all parties hold a random additive sharing $[\Delta]$, where Δ is a global MAC key. An authenticated sharing $[\![x]\!]$ is defined by three additive sharings $[\![x]\!] := ([x], [\Delta], [\Delta \cdot x])$. Later on, when parties open the secret x, they can reconstruct Δ and $\Delta \cdot x$ to verify the correctness of x. Intuitively, if an adversary wants honest parties to accept an incorrect $x' \neq x$, he needs to correctly guess the MAC key Δ, which happens with negligible probability.

Now in the preprocessing phase, all parties prepare $|C|$ authenticated Beaver triples in the form of $([\![a]\!], [\![b]\!], [\![c]\!])$. In the online phase, parties first run the semi-honest SPDZ protocol and then use MACs to verify the correctness of all public reconstruction. In particular, the verification can be done with sublinear communication by checking a random linear combination of all opened authenticated sharings. Thus, the amortized communication per multiplication gate remains to be $4n$ field elements.

Preparing Authenticated Triples. To maintain the amortized communication of $4n$ elements per multiplication gate, the only task is to prepare authenticated Beaver triples with sublinear communication in the offline phase. Unfortunately, it is not known in the literature how this can be achieved, unless one allows a large additional communication overhead of $O(\sqrt{|C|} \cdot n^3)$ bits [AS22].

In [RS22], the authors construct a PCG which can efficiently generate *partially* authenticated Beaver triples in the form of $(\llbracket a \rrbracket, \llbracket b \rrbracket, [c])$, i.e., only the first two additive sharings in each triple are authenticated. To obtain authenticated Beaver triples, the c sharing is authenticated by consuming a random authenticated sharing $\llbracket r \rrbracket$. In short, parties publicly reconstruct $[c + r]$ and compute $\llbracket c \rrbracket := (c + r) - \llbracket r \rrbracket$. The public reconstruction costs $2n$ elements of communication.

Making it even worse, the PCG in [RS22] does not ensure the correct correlation: an adversary may insert linear errors on c by choosing arbitrary constants $\delta_a, \delta_b, \delta_c$ and causing $c = a \cdot b + \delta_a \cdot a + \delta_b \cdot b + \delta_c$. As a result, after authenticating the c sharing, parties also need to verify the correctness of the authenticated Beaver triple, which can be done by using the sacrifice technique [DPSZ12]. This requires consuming another (possibly incorrect) authenticated Beaver triple with public reconstruction of 2 authenticated additive sharings. In total, the overall communication complexity becomes $12n$ elements per multiplication gate in [RS22]. This is improved implicitly in [GLM+24] to $10n$ elements per multiplication gate with a better sacrifice technique [KOS16].

3.2 Generating Partially Authenticated Beaver Triples without Linear Errors

Our idea starts from the following key observation of the PCG protocol in [RS22]. Let k be the number of partially authenticated Beaver triples produced by the PCG. During the computation, all parties obtain not only $\{(\llbracket a_i \rrbracket, \llbracket b_i \rrbracket, [c_i])\}_{i=1}^k$, but also $[a_i \cdot b_j]$ for all $i, j \in \{1, 2, \ldots, k\}$ as by-products. We refer to this type of correlated randomness $(\{\llbracket a_i \rrbracket\}_{i=1}^k, \{\llbracket b_j \rrbracket\}_{j=1}^k, \{[a_i \cdot b_j]\}_{i,j=1}^k)$ as a length-k partially authenticated tensor product.

These additional sharings of $a_i \cdot b_j$ enable a simpler verification of multiplication correlations. The main benefit of having tensor-product correlation is that for all a that is a linear combination of $\{a_1, \ldots, a_k\}$ and for all b that is a linear combination of $\{b_1, \ldots, b_k\}$, all parties can locally compute an additive secret sharing $[a \cdot b]$ from $\{[a_i \cdot b_j]\}_{i,j=1}^k)$. To be more concrete, for all public vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}^k$, let $\llbracket a \rrbracket = \sum_{i=1}^k u_i \cdot \llbracket a_i \rrbracket$ and $\llbracket b \rrbracket = \sum_{i=1}^k v_i \cdot \llbracket b_i \rrbracket$. All parties can locally compute

$$[a \cdot b] := \sum_{i=1}^{k} \sum_{j=1}^{k} u_i \cdot v_j \cdot [a_i \cdot b_j].$$

Now, to verify the correctness of the multiplication correlation for each (a_i, b_i, c_i), all parties do the following steps.

1. Generate random public vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}^k$ and locally compute $[a \cdot b]$.
2. Convert $[a \cdot b]$ to $\llbracket a \cdot b \rrbracket$ by consuming a random authenticated sharing $\llbracket r \rrbracket$.
3. Robustly open $\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket a \cdot b \rrbracket$ and verify the multiplication correlation of the secrets.

Note that the communication complexity of the above verification is sublinear in k.

To see why this check is effective, recall that during the computation of the PCG in [RS22], an adversary may insert linear errors on c_i. In fact, it may insert linear errors for every $[a_i \cdot b_j]$ by choosing $\delta_{i,j}^{(a)}, \delta_{i,j}^{(b)}$ and causing the secret of $[a_i \cdot b_j]$ to be $a_i \cdot b_j + \delta_{i,j}^{(a)} \cdot a_i + \delta_{i,j}^{(b)} \cdot b_j$. (Here we omit the additive error on the secret since for an additive sharing, corrupted parties can insert additive errors at any time by changing their shares locally. Our verification only ensures that there is no linear error on c_i.) As a result, the secret of $\llbracket a \cdot b \rrbracket$ becomes

$$a \cdot b + \sum_{i=1}^{k} a_i \cdot \left(\sum_{j=1}^{k} u_i v_j \delta_{i,j}^{(a)} \right) + \sum_{j=1}^{k} b_j \cdot \left(\sum_{i=1}^{k} u_i v_j \delta_{i,j}^{(b)} \right) + \delta^{(c)}$$

for some additive error $\delta^{(c)}$ chosen by the adversary as well. Note that the coefficients of a_i's and b_j's in the above equation are known to the adversary. The key observation is that, when $\boldsymbol{u}$ and $\boldsymbol{v}$ are randomly

chosen, if at least one of $\delta_{i,j}^{(a)}$ or $\delta_{i,j}^{(b)}$ is not zero, with overwhelming probability, the secrets of $[\![a]\!], [\![b]\!], [\![a \cdot b]\!]$ do not satisfy the correct multiplication correlation.

As a result, if the verification passes, all parties hold a correct length-k tensor product (without linear errors), which also implies the correct multiplication correlation for each (a_i, b_i, c_i) (without linear errors).

Remark 1 *Unfortunately, the above verification is not useful for the protocol in [RS22]. Indeed, an additive secret sharing without authentication cannot prevent corrupted parties from inserting an additive error in the secret. When authenticating the c sharing to obtain an authenticated Beaver triple, the adversary still has a chance to insert an additive error on c, leading to incorrect multiplication correlation again. As a result, parties still need to verify the authenticated Beaver triples as in [RS22].*

However, we will show that such a verification is important in our construction. Jumping ahead, we will directly use partially authenticated Beaver triples in the online evaluation phase, without authenticating the c sharing. The above verification ensures that an adversary can only insert additive errors in the online phase, so that we can apply the recursive check in [GS20] to verify the online computation with sublinear communication.

General Approach for Partially Authenticated Tensor Products. We observe that length-k partially authenticated tensor products can be prepared using essentially any existing PCG construction for random Beaver triples in the literature, rather than being restricted to the PCG of [RS22]. Motivated by this observation, we present a more general approach for preparing partially authenticated tensor products.

In particular, we show that length-k partially authenticated tensor products can be derived from length-$(k+1)$ unauthenticated tensor products in the form of

$$(\{[a_i]\}_{i=1}^{k+1}, \{[b_j]\}_{j=1}^{k+1}, \{[a_i \cdot b_j]\}_{i,j=1}^{k+1}).$$

The key observation is that, since the parties hold $\{[a_{k+1} \cdot b_j]\}_{j=1}^{k}$, the value a_{k+1} can be viewed as a local MAC key for $\{[b_j]\}_{j=1}^{k}$. Meanwhile, the parties also possess an additive sharing $[\Delta]$ of the global MAC key Δ. Therefore, they can transform $\{[a_{k+1} \cdot b_j]\}_{j=1}^{k}$ into $\{[\Delta \cdot b_j]\}_{j=1}^{k}$, thereby obtaining $\{[\![b_j]\!]\}_{j=1}^{k}$, as follows:

1. Locally compute $[\Delta + a_{k+1}]$ and open the resulting secret.
2. For each $j = 1, \ldots, k$, locally compute $[\Delta \cdot b_j] = (\Delta + a_{k+1}) \cdot [b_j] - [a_{k+1} \cdot b_j]$.

In the above process, the adversary can only introduce an additive error into the opening of $[\Delta + a_{k+1}]$, which in turn induces a linear error in each $[\Delta \cdot b_j]$. Since each b_j is uniformly random, any non-zero linear error results in a uniformly random error on the corresponding MAC share $[\Delta \cdot b_j]$. Consequently, it suffices to open a random linear combination A of all the $[b_j]$'s and verify its MAC. Because the error on at least one $[\Delta \cdot b_j]$ is random, the induced error on $[\Delta \cdot A]$ is also random. To pass the MAC check, the adversary must additionally introduce a matching additive error into the corrupted parties' shares of $[A]$, which succeeds only with negligible probability. Hence, except with negligible probability, any cheating attempt causes the honest parties to abort.

Similarly, we can also convert $\{[b_{k+1} \cdot a_i]\}_{i=1}^{k}$ to $\{[\Delta \cdot a_i]\}_{i=1}^{k}$ and then obtain a desired length-k partially authenticated tensor product correlation. During the whole process, only a constant number of field elements need to be opened, which is quite communication-efficient.

Now, the remaining problem is how to prepare length-$(k+1)$ (n-party) unauthenticated tensor products. In particular, we show that such correlations can be prepared directly from either of the two simpler correlations:

- Programmable 2-party tensor product correlations (see Section 4.3);
- Programmable n-party unauthenticated triples [BCG+20].

Importantly, both of these correlations are already implicit in, or can be directly obtained from, existing PCG constructions for Beaver triples in the literature. In particular, programmable 2-party tensor product

correlations naturally arise in LPN-based PCGs for OLE correlations, which serve as essential building blocks in existing PCGs for Beaver triples. Unauthenticated triples are precisely the correlations required by semi-honest SPDZ protocols, where the programmability is also satisfied by most of the existing constructions. We refer the readers to Section 4 for more discussions.

3.3 Online Evaluation with Partially Authenticated Beaver Triples

To further avoid the cost of authenticating the c sharing in each Beaver triple, our idea is to directly use partially authenticated Beaver triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, [c])$ in the online computation. We note that for each multiplication gate with input sharings $[x], [y]$, after all parties reconstruct $x + a$ and $y + b$, they can obtain *authenticated* sharings for the inputs $\llbracket x \rrbracket := (x + a) - \llbracket a \rrbracket$ and $\llbracket y \rrbracket := (y + b) - \llbracket b \rrbracket$ while the output $[z]$ is only additively shared. As a result, after the evaluation, all parties will obtain authenticated sharings for the input wires of each multiplication gate.

We note that for each multiplication gate, when reconstructing $x + a$ from $[x + a]$, the adversary may insert an additive error on the secret. In [DKL+13], such an attack can be detected relying on authenticated sharings for all wires in the circuit. I.e., parties hold $\llbracket x \rrbracket$ and $\llbracket a \rrbracket$ *before* evaluating this multiplication gate, and it is sufficient to verify that $x+a$ is the correct opening of $\llbracket x \rrbracket + \llbracket a \rrbracket$. In our case, however, since $[c]$ in the Beaver triple is not authenticated, parties may not hold $\llbracket x \rrbracket$. For example, x might be an output wire of a multiplication gate in the last layer, where parties only hold an additive sharing of x without authentication. As a result, we cannot verify the computation as [DKL+13].

To verify the computation, we will check that all authenticated wire values satisfy the correct correlations defined by the circuit. To be more concrete, suppose (x_i, y_i) are the input wires of the i-th multiplication gate which have been authenticated during the computation. Then x_i is computed as a suitable linear combination of the outputs of the previous $i - 1$ multiplication gates, and each such output z_j (with $j < i$) can be further represented as the multiplication of the two authenticated inputs $x_j \cdot y_j$. Thus, we obtain the following inner-product restriction for x_i:

$$x_i = \sum_{j=1}^{i-1} \alpha_i^{(j)} \cdot x_j \cdot y_j,$$

where $\{\alpha_i^{(j)}\}_{j=1}^{i-1}$ are constants determined by the circuit structure, and parties hold authenticated sharings for each variable in the equation. Thus, the verification of the computation boils down to checking $O(|C|)$ inner-product tuples, one for every input wire of each multiplication gate.

In addition, thanks to the verification of the partially authenticated Beaver triples in the preprocessing phase, what an adversary can do in the online phase is restricted to inserting an additive error on each x_i. As a result, the difference $x_i - \sum_{j=1}^{i-1} \alpha_i^{(j)} \cdot (x_j \cdot y_j)$ (which should be 0 if the computation is correct) is known to the adversary.

Inner-Product Verification. To perform the verification efficiently, all parties check a random linear combination of all inner-product tuples, which becomes a single inner product of size $|C|$:

$$z = \sum_{i=1}^{|C|} \alpha_i \cdot (x_i \cdot y_i),$$

where α_i is a public coefficient determined by the circuit structure and the random challenge for the linear combination. In addition, this inner-product tuple is only subject to additive attack where the adversary knows $z - \sum_{i=1}^{|C|} \alpha_i \cdot (x_i \cdot y_i)$.

Our idea is to follow the recursive check in [GS20] for inner-product relation subject to additive attacks, which only incurs a sublinear communication in the size of input vectors. However, the original technique in [GS20] is designed for values shared by degree-t Shamir sharings in the honest majority setting. In particular, the communication efficiency of their verification heavily relies on the fact that computing the

inner product of two shared vectors costs the same as one single multiplication, independent of the size of the input vectors. This is because with degree-t Shamir sharings and honest majority where $n = 2t + 1$, all parties can first locally compute a degree-$2t$ Shamir sharing of the inner-product result, and then apply the DN technique [DN07] to reduce the degree to t. This unfortunately does not work in the dishonest majority setting: To compute the inner product of two vectors of authenticated sharings, parties need to compute each multiplication using a fresh Beaver triple and then sum up all multiplication results. This leads to linear communication in the input vector size of the inner product.

To overcome this issue, we rely on the length-k partially authenticated tensor product again. For simplicity, let's assume that $k = |C|$, i.e., all random Beaver triples are generated from one instance of PCG. The key insight is that when applying the recursive check in [GS20], each inner product with input vectors $\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}}$ satisfy that every element in $\tilde{\boldsymbol{x}}$ is a linear combination of $\{x_1, \ldots, x_{|C|}\}$, i.e., the first input wires of all multiplication gates, and every element in $\tilde{\boldsymbol{y}}$ is a linear combination of $\{y_1, \ldots, y_{|C|}\}$, i.e., the second input wires of all multiplication gates. Therefore, $\langle \tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}} \rangle$ can be computed by a linear combination of $\{x_i \cdot y_j\}_{i,j=1}^{|C|}$. Note that

- During the evaluation, parties have reconstructed $x_i + a_i$ and $y_i + b_i$ when evaluating the i-th multiplication gate using the Beaver triple $(\llbracket a_i \rrbracket, \llbracket b_i \rrbracket, [c_i])$;
- Parties also hold $[a_i \cdot b_j]$ for all $i, j \in \{1, \ldots, |C|\}$.

Thus, for every $i, j \in \{1, \ldots, |C|\}$, all parties can locally compute

$$[x_i \cdot y_j] = (x_i + a_i)(y_j + b_j) - (x_i + a_i)[b_j] - (y_j + b_j)[a_i] + [a_i \cdot b_j].$$

As a result, all parties can further locally compute an additive sharing $[\langle \tilde{\boldsymbol{x}}, \tilde{\boldsymbol{y}} \rangle]$ and then authenticate it via a random authenticated sharing $\llbracket r \rrbracket$. In this way, the cost of each inner product operation required by the recursive check in [GS20] is also independent of the vector size. We refer the readers to Section 6 for the formal treatment of the verification.

Optimizing Computational Complexity. The above verification assumes a length-$|C|$ tensor product. While efficient in communication, it leads to a quadratic *computational complexity* in the circuit size! We note that it is sufficient to use length-k tensor products, at the cost of an additional $O(|C| n \cdot \frac{\log k}{k})$ elements of communication. At a high level, we will divide the single inner-product tuple to be checked into $|C|/k$ inner-product tuples of size k, such that each small inner-product tuple only involves Beaver triples from the same length-k tensor product. Then we may apply the same idea to verify each small inner-product tuple separately.

Notably, this does not mean that the $\{[a_i]\}_{i=1}^{k}$ (similarly for $\{[b_i]\}_{i=1}^{k}$) across all the tensor products needs to be generated from $|C|/k$ independent seeds (which results in additional overhead on the total key size). We may regard the $|C|/k$ tensor product correlations of length-k as a single correlation, where each party's the pseudorandom shares of $\{[a_i]\}_{i=1}^{k}$ (similarly for $\{[b_i]\}_{i=1}^{k}$) across all the tensor products can still be generated from a single seed as in the semi-honest SPDZ protocol. We refer the readers to Section 4 for more details.

3.4 Protocol Summary and Further Optimization

We summarize our construction as follows. For simplicity, we assume the circuit C only contains public outputs. Let m_I/m_O be the number of input/output wires in C and $k = \omega(1)$.

1. **Preprocessing Phase.** All parties prepare $m_O + m_I + O(|C| \cdot \frac{\log k}{k})$ random authenticated sharings and $|C|/k$ length-$(k+1)$ (unauthenticated) tensor products. Then, the parties will convert the length-$(k+1)$ tensor products to length k partially authenticated tensor products, which implies $|C|$ partially authenticated Beaver triples.

   For each length-k partially authenticated tensor product, all parties run the verification protocol in Section 3.2 to ensure that there are no linear errors. (The verification, in fact, would sacrifice one

random Beaver triple. In our construction, parties will prepare $O(|C|/k^2)$ additional tensor products that are sacrificed.)

2. **Online Phase.** Parties first authenticate each input wire with a random authenticated sharing. Then all parties evaluate the circuit gate by gate using semi-honest SPDZ online protocol. Finally, the parties use m_O random authenticated sharings to authenticate the output wires. For each input wire value x of a multiplication gate, a masked value $x + a$ has been opened with $[\![a]\!]$ held by the parties. Thus, x has been authenticated by $[\![x]\!] = (x + a) - [\![a]\!]$.

3. **Verification Phase.** All parties determine $O(|C|)$ inner-product restrictions, one for each authenticated wire. Then all parties generate random coefficients and compute a random linear combination of all inner-product restrictions. As a result, the verification is reduced to check a single inner-product tuple of size $|C|$.

   All parties divide this single inner-product tuple into $|C|/k$ small inner-product tuples such that each inner-product tuple only involves Beaver triples from the same length-k partially authenticated tensor product. Then all parties follow [GS20] to verify each small inner-product with sublinear communication.

4. **Output Phase.** Upon successful verification, the parties robustly reconstruct the secret value for each output of C.

Turbospeedz Optimization. We note that utilizing a variant of length-$|C|$ partially authenticated tensor product, we can further reduce the communication complexity per gate to just $2n$ field elements (without applying the optimization of quadratic computational complexity in Section 3.3), following the online protocol from Turbospeedz [BNO19]. We briefly outline this optimization here.

In this approach, parties will only compute an authenticated sharing for each input wire of the circuit and the *output wire* of each multiplication gate. In this way, when evaluating a multiplication gate, parties may compute authenticated input sharings $[\![x]\!], [\![y]\!]$. Then they will use the length-$|C|$ partially authenticated tensor product to compute an additive sharing of the output, which is then authenticated via a random authenticated sharing.

With more details, all parties will prepare circuit-independent preprocessing data in the form of $(\{[a_i]\}_{i=1}^{|C|+1}, \{[a_i \cdot a_j]\}_{i,j=1}^{|C|+1})$ and use it to compute $(\{[\![a_i]\!]\}_{i=1}^{|C|}, \{[a_i \cdot a_j]\}_{i,j=1}^{|C|})$. Effectively, this is a length-$|C|$ partially authenticated tensor product with $\boldsymbol{a} = \boldsymbol{b}$. Such a correlation can also be prepared using the techniques introduced in Section 3.2. Suppose all input wires of the circuit and output wires of all multiplication gates are denoted by $z_1, z_2, \ldots, z_{|C|}$. In the online phase, parties will compute an authenticated sharing $[\![z_i]\!]$ for every $i \in \{1, \ldots, |C|\}$ as follows.

1. All parties first authenticate each input wire z_i with $[\![a_i]\!]$. As a result, all parties hold $[\![z_i]\!] = (z_i + a_i) - [\![a_i]\!]$.

2. For each multiplication gate with input wires x_i, y_i, all parties first compute $[\![x_i]\!], [\![y_i]\!]$, which are linear combinations of $\{[\![z_j]\!]\}_{j<i}$ computed in previous layers.

3. Note that $x_i \cdot y_i$ can be computed as a linear combination of $\{z_{j_1} \cdot z_{j_2}\}_{j_1,j_2=1}^{|C|}$. Also note that all parties can locally compute $[z_{j_1} \cdot z_{j_2}]$ using public values $z_{j_1} + a_{j_1}, z_{j_2} + a_{j_2}$ and additive sharings $[a_{j_1}], [a_{j_2}], [a_{j_1} \cdot a_{j_2}]$. Thus, all parties locally compute an additive sharing $[x_i \cdot y_i]$ and authenticate it using $[\![z_i]\!]$. As a result, all parties hold $[\![z_i]\!] = (z_i + a_i) - [\![a_i]\!]$.

Following the same analysis, after the verification in the preprocessing phase, an adversary can only launch additive attacks in the online phase. This time, parties can compute authenticated sharings for all wires in the circuits. Thus, the verification is reduced to checking the multiplication relation for each multiplication gate. We may again convert it to the check of a single inner-product of size $|C|$, and then apply the recursive check in [GS20]. For further details, we refer the reader to Section 8.

4 Generating Partially Authenticated Tensor Products

In this section, we introduce the generation of preprocessing data required in our protocol, which mainly consists of length-k (n-party) partially authenticated tensor products.

4.1 Functionality for Partially Authenticated Tensor Products

We first give the functionality for the length-k partially authenticated tensor products that we required for our preprocessing. The functionality is an n-party functionality, providing an additive sharing $[\Delta]$, where Δ serves as the global MAC key, and m_T partially authenticated triples in the form of $\{[\![a_i]\!], [\![b_i]\!]\}_{i=1}^k, \{[c_{i,j}]\}_{i,j=1}^k$ with $c_{i,j} = a_i \cdot b_j$. The functionality $\mathcal{F}_{\mathsf{patensor}}^k$ is given in Figure 1.

Functionality $\mathcal{F}_{\mathsf{patensor}}^k$

On receiving (PATens, m_T) from all the honest parties:

1. The trusted party samples a random element in $\mathbb{F}$ as Δ.
2. The trusted party receives corrupted parties' shares of $[\Delta]$ from Sim and samples the honest parties' shares of $[\Delta]$ based on Δ and the corrupted parties' shares of $[\Delta]$.
3. Repeat the following m_T times:
   (a) The trusted party samples field elements $\{a_i, b_i\}_{i=1}^k$ randomly and computes $\Delta \cdot a_i, \Delta \cdot b_i$ for each $i = 1, \ldots, k$.
   (b) The trusted party receives corrupted parties' shares of $\{[\![a_i]\!], [\![b_i]\!]\}_{i=1}^k, \{[c_{i,j}]\}_{i,j=1}^k$ from Sim. Then, the trusted party samples the honest parties' shares of $\{[\![a_i]\!], [\![b_i]\!]\}_{i=1}^k$ based on the secrets and corrupted parties' shares.
   (c) For each pair of $(i,j) \in \{1, \ldots, k\}^2$, the trusted party computes $c_{i,j} = a_i \cdot b_j$ and receives corrupted parties' shares of $[c_{i,j}]$ from Sim. Then, the trusted party samples the honest parties' shares of $[c_{i,j}]$ based on the secret and the corrupted parties' shares.
   (d) The trusted party distributes $\{([\![a_i]\!], [\![b_i]\!])\}_{i=1}^k, \{[c_{i,j}]\}_{i,j=1}^k$ to the parties.

Abort: If (abort, P) is received from Sim for an honest party P, the trusted party sends abort to P.

Figure 1: Functionality of length-k n-party partially authenticated tensor products.

4.2 Construction from Unauthenticated Tensor Products

We show that the partially authenticated tensor products can be constructed from unauthenticated tensor products in a black-box way. We first give the description of the functionality $\mathcal{F}_{\mathsf{uatensor}}^k$ that models unauthenticated length-k n-party tensor product correlations. In particular, each tensor product correlation contains $\{[a_i], [b_i]\}_{i=1}^k, \{[c_{i,j}]\}_{i,j=1}^k$ with $c_{i,j} = a_i \cdot b_j$. We allow linear errors in a_i, b_j on each $[c_{i,j}]$.

Functionality $\mathcal{F}_{\mathsf{uatensor}}^k$

On receiving (UATens, m_U) from all the honest parties, repeat the following m_U times:

1. The trusted party samples field elements $\{a_i, b_i\}_{i=1}^k$ randomly.
2. The trusted party receives corrupted parties' shares of $\{[a_i], [b_i]\}_{i=1}^k, \{[c_{i,j}]\}_{i,j=1}^k$ from Sim. Then, the trusted party samples the honest parties' shares of $\{[a_i], [b_i]\}_{i=1}^k$ based on the secrets and corrupted parties' shares.
3. For each pair of $(i,j) \in \{1, \ldots, k\}^2$, the trusted party receives $\delta_{a,i,j}, \delta_{b,i,j}$ from Sim. The trusted party sets $\delta_{b,1,i} = \delta_{a,i,1} = 0$ for each $i = 1, \ldots, k$ and then computes $c_{i,j} = a_i \cdot b_j + \delta_{a,i,j} \cdot b_j + \delta_{b,i,j} \cdot a_i$ for each pair of $(i,j) \in \{1, \ldots, k\}^2$.
4. The trusted party samples the honest parties' shares of $[c_{i,j}]$ based on the secret and the corrupted parties' shares.

5. The trusted party distributes $\{([a_i],[b_i])\}_{i=1}^k, \{[c_{i,j}]\}_{i,j=1}^k$ to the parties.

Abort: If (abort, P) is received from Sim for an honest party P, the trusted party sends abort to P.

Figure 2: Functionality of length-k n-party unauthenticated tensor products.

Now, we give the protocol $\Pi^k_{\mathsf{patensor}}$ in the $\{\mathcal{F}^{k+1}_{\mathsf{uatensor}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model that instantiates $\mathcal{F}^k_{\mathsf{patensor}}$. The security proof can be found in Section C.

Protocol $\Pi^k_{\mathsf{patensor}}$

Public Parameter: The number m_T of partially authenticated tensor products we need to generate.

1. The parties locally sample their shares of $[\Delta]$.
2. The parties send $(\mathsf{UATens}, m_T + \lceil m_T/k \rceil + 1)$ to $\mathcal{F}^{k+1}_{\mathsf{uatensor}}$ and receives $m_T + \lceil m_T/k \rceil + 1$ unauthenticated tensor products. In the α-th tensor product, we use $\Delta_a^{(\alpha)}, \Delta_b^{(\alpha)}$ to denote the elements a_1, b_1 in $\mathcal{F}^{k+1}_{\mathsf{uatensor}}$, and then use $a_i^{(\alpha)}, b_i^{(\alpha)}$ to denote the elements a_{i+1}, b_{i+1} in $\mathcal{F}^{k+1}_{\mathsf{uatensor}}$, respectively. In other words, the tensor products are denoted by $[\Delta_a^{(\alpha)}], [\Delta_b^{(\alpha)}]$,
$\{[a_i^{(\alpha)}], [b_i^{(\alpha)}], [d_i^{(\alpha)} = (\Delta_a^{(\alpha)} + \delta_{a,i}^{(\alpha)}) \cdot b_i^{(\alpha)}], [e_i^{(\alpha)} = (\Delta_b^{(\alpha)} + \delta_{b,i}^{(\alpha)}) \cdot a_i^{(\alpha)}]\}_{i=1}^k$, and
$\{[c_{i,j}^{(\alpha)} = a_i^{(\alpha)} \cdot b_j^{(\alpha)} + \delta_{a,i,j}^{(\alpha)} \cdot b_j^{(\alpha)} + \delta_{b,i,j}^{(\alpha)} \cdot a_i^{(\alpha)}]\}_{i,j=1}^k$ for $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$.
3. For each $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$, the parties locally compute $[\Delta + \Delta_a^{(\alpha)}], [\Delta + \Delta_b^{(\alpha)}]$ and sends them to P_{king}. P_{king} reconstructs $\Delta + \Delta_a^{(\alpha)}, \Delta + \Delta_b^{(\alpha)}$ and sends them to all the parties.
4. For each $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$, the parties locally compute $[\Delta \cdot a_i^{(\alpha)}] = (\Delta + \Delta_b^{(\alpha)}) \cdot [a_i^{(\alpha)}] - [e_i^{(\alpha)}], [\Delta \cdot b_i^{(\alpha)}] = (\Delta + \Delta_a^{(\alpha)}) \cdot [b_i^{(\alpha)}] - [d_i^{(\alpha)}]$ for each $i = 1, \ldots, k$. Then, the parties hold $\{[\![a_i^{(\alpha)}]\!], [\![b_i^{(\alpha)}]\!]\}_{i=1}^k$ for each $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$.
5. The parties send $\mathsf{RandCoin}$ to $\mathcal{F}_{\mathsf{Coin}}$ and obtain $r \in \mathbb{F}^*$. Let $\mathsf{PRG} : \mathbb{F}^* \to \mathbb{F}^{2k(m_T + \lceil m_T/k \rceil + 1)}$ be a public pseudorandom generator. Then, let

$$\mathsf{PRG}(r) = (\{r_{a,1}^{(\alpha)}, \ldots, r_{a,k}^{(\alpha)}, r_{b,1}^{(\alpha)}, \ldots, r_{b,k}^{(\alpha)}\}_{\alpha=1}^{m_T + \lceil m_T/k \rceil + 1}),$$

and the parties locally compute $[\![A]\!] = \sum_{\alpha=1}^{m_T + \lceil m_T/k \rceil + 1} \sum_{i=1}^k (r_{a,i}^{(\alpha)} [\![a_i^{(\alpha)}]\!] + r_{b,i}^{(\alpha)} [\![b_i^{(\alpha)}]\!])$.
6. The parties run Π_{Open} on $[\![A]\!]$ and then run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on the opened value A. If the check passes, the parties now have $\{([\![a_i^{(\alpha)}]\!], [\![b_i]\!]^{(\alpha)})\}_{i=1}^k, \{[c_{i,j}^{(\alpha)}]\}_{i,j=1}^k$ for $\alpha = 1, \ldots, 1, \ldots, m_T + \lceil m_T/k \rceil$ (where the last tensor product is sacrificed during the verification in the last step). The tensor products for $\alpha = m_T + 1, \ldots, m_T + \lceil m_T/k \rceil$ are regarded as m_T partially authenticated triples $\{([\![x'_\alpha]\!], [\![y'_\alpha]\!], [v'_\alpha = x'_\alpha \cdot y'_\alpha + \delta_x^{(\alpha)} \cdot y'_\alpha + \delta_y^{(\alpha)} \cdot x'_\alpha])\}_{\alpha=1}^{m_T}$, which are used for checking the first m_T tensor products.
7. The parties send $\mathsf{RandCoin}$ to $\mathcal{F}_{\mathsf{Coin}}$ and receive a random element $r' \in \mathbb{F}^*$ from it. Then, the parties locally compute $\mathsf{PRG}(r')$ and extract the first $2km_T$ field elements as

$$\{(r_{a,\alpha}^{(1)'}, \ldots, r_{a,\alpha}^{(k)'}, r_{b,\alpha}^{(1)'}, \ldots, r_{b,\alpha}^{(k)'})\}_{\alpha=1}^{m_T}.$$

For each $\alpha = 1, \ldots, m_T$:

(a) The parties locally compute $[\![x_\alpha]\!] = \sum_{i=1}^k r_{a,\alpha}^{(i)'} [\![a_i^{(\alpha)}]\!]$ and $[\![y_\alpha]\!] = \sum_{i=1}^k r_{b,\alpha}^{(i)'} [\![b_i^{(\alpha)}]\!]$. Then, the parties locally compute

$$[v_\alpha] = \sum_{i=1}^k \sum_{j=1}^k r_{a,\alpha}^{(i)'} r_{b,\alpha}^{(j)'} [c_{i,j}^{(\alpha)}].$$

(b) The parties locally compute $[\![x_\alpha + x'_\alpha]\!], [\![y_\alpha + y'_\alpha]\!]$ and run Π_{Open} on them. Then, the parties locally compute

$$[\tau^{(\alpha)}] = (x_\alpha + x'_\alpha) \cdot (y_\alpha + y'_\alpha) - (x_\alpha + x'_\alpha) \cdot [y'_\alpha] - (y_\alpha + y'_\alpha) \cdot [x'_\alpha] + [v'_\alpha] - [v_\alpha].$$

After computing $\{[\tau^{(\alpha)}]\}_{\alpha=1}^{m_T}$, the parties compute $[\tau] = \sum_{\alpha=1}^{m_T}[\tau^{(\alpha)}]$. Then, the parties run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on all the opened values during the verification, i.e., $\{x_\alpha + x'_\alpha, y_\alpha + y'_\alpha\}_{\alpha=1}^{m_T}$. Finally, each party P_i calls $\mathcal{F}_{\mathsf{Commit}}$ with input $(\mathsf{commit}, P_i, \tau_i, \mathsf{mid}_{\tau_i})$, where τ_i is P_i's share of $[\tau]$. The parties then open their commitments via sending $(\mathsf{open}, P_i, \mathsf{mid}_{\tau_i})$ for each party P_i and check that $\sum_{i=1}^{n} \tau_i = 0$. If not, the parties abort the protocol.

Figure 3: Generating partially authenticated tensor products from unauthenticated tensor products.

Except for the invocations of functionalities, only Steps 3, 6, and 7 require communication. In Step 3, the parties communicate $O(m_T n)$ elements. In Step 6, the parties only open one element. In Step 7, to verify each length-k tensor product, the servers need to run Π_{Open} on 2 elements, which requires $4n$ communication. Thus, the total communication cost is $O(m_T n)$.

4.3 Instantiation of Unauthenticated Tensor Products

Now we focus on the instantiations of $\mathcal{F}_{\mathsf{uatensor}}^k$ that produce unauthenticated tensor products.

Instantiation from 2-Party Tensor Products. First, we show that the unauthenticated tensor products can be generated from (programmable) 2-party tensor product correlations, which is modeled as a functionality that generates an additive sharing of the tensor product of two vectors $\boldsymbol{u}, \boldsymbol{x}$ in $\mathbb{F}^k$. The *programmability* requires that, for any given instance of the functionality, the party obtaining $\boldsymbol{u}$ or $\boldsymbol{x}$ can program them from a chosen random seed. This allows the same $\boldsymbol{u}, \boldsymbol{x}$ to be used in different instances of $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}$. The programmability is modeled with an expansion function $\mathsf{Expand} : S \to \mathbb{F}^m$, which generates $\ell = m/k$ blocks of tensor product correlations together. $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}$ is provided below in Figure 4.

Functionality $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m, k)$

Let $\mathsf{Expand}_{m,k} : S \to \mathbb{F}^m$ be an expansion function with seed space S and output length m, parameterized by a factor k. Let $\ell = m/k$. On receiving $s_a \in S$ from P_A and $s_b \in S$ from P_B:

1. The trusted party computes $\boldsymbol{u} = \mathsf{Expand}_{m,k}(s_a), \boldsymbol{x} = \mathsf{Expand}_{m,k}(s_b)$. Let $\boldsymbol{u} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_\ell), \boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\ell) \in (\mathbb{F}^k)^\ell$. The trusted party then samples $\boldsymbol{v}_i \in \mathbb{F}^{k^2}$ for $i = 1, \ldots, \ell$.
2. For each $i = 1, \ldots, \ell$, the trusted party outputs $\boldsymbol{w}_i = \boldsymbol{u}_i \otimes \boldsymbol{x}_i + \boldsymbol{v}_i$ to P_A and $\boldsymbol{v}_i$ to P_B.

Corrupted Parties: If P_B is corrupted, each $\boldsymbol{v}_i$ is chosen by Sim. For a corrupt P_A, Sim can choose $\boldsymbol{w}_i$ (and then $\boldsymbol{v}$ is recomputed accordingly). If either of the parties is corrupted, Sim can send abort to the trusted party, and then the trusted party will send abort to the other party.

Figure 4: Functionality of programmable PCG for tensor product correlation.

Remark 2 *Note that a typical way of instantiating the functionality is from LPN-type assumptions, where the approach of using $\ell = m/k$ independent seeds to expand to $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_\ell$ (and similar for $\boldsymbol{x}$) has the lowest computational cost. Nevertheless, we do not restrict the expansion algorithm to have such a specific form in general. More details can be found in Section 4.4.*

Then, we provide the protocol $\Pi_{\mathsf{uatensor}}^k$ for instantiating $\mathcal{F}_{\mathsf{uatensor}}^k$ in the $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$-hybrid model below in Figure 5.

Protocol $\Pi_{\mathsf{uatensor}}^k$

Public Parameter: The number m_U of unauthenticated tensor products we need to generate.

1. Each party P_i samples two random seeds $s_a^{(i)}, s_b^{(i)}$ and computes $\boldsymbol{a}^{(i)} = \mathsf{Expand}_{m_U \cdot k, k}(s_a^{(i)}), \boldsymbol{b}^{(i)} = \mathsf{Expand}_{m_U \cdot k, k}(s_b^{(i)})$ as his shares of $\{[a_i]\}_{i=1}^k, \{[b_i]\}_{i=1}^k$ for all m_U tensor products (denoted by $[\boldsymbol{a}], [\boldsymbol{b}]$ for simplicity), respectively.
2. For each (ordered) pair of parties (P_i, P_j), P_i sends $s_a^{(i)}$ and P_j sends $s_b^{(j)}$ to $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$. If abort is

received, the party aborts the protocol. Otherwise, P_i receives $\boldsymbol{w}_{i,j}$ and P_j receives $\boldsymbol{v}_{i,j}$ from $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$.

3. To get the α-th ($1 \leq \alpha \leq m_U$) tensor product, let $\boldsymbol{a}_\alpha, \boldsymbol{b}_\alpha$ be the vectors of the $k^2(\alpha-1)+1, \ldots, k^2\alpha$-th entries of $\boldsymbol{a}, \boldsymbol{b}$ respectively. Each party P_i computes $\boldsymbol{a}^{(i)} \otimes \boldsymbol{b}^{(i)} + \sum_{j \neq i}(\boldsymbol{w}_{i,j} - \boldsymbol{v}_{j,i})$ and extracts the $k^2(\alpha-1)+1, \ldots, k^2\alpha$-th entries of it as his share of $[\boldsymbol{a}_\alpha \otimes \boldsymbol{b}_\alpha]$.

Figure 5: Generating unauthenticated tensor products from 2-party tensor product correlations.

The security proof is provided in Section D.1. The protocol requires $O(n^2)$ invocations of $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m, k)$ and no additional communication.

Remark 3 *As we showed in the security proof, this approach of realizing $\mathcal{F}_{\mathsf{uatensor}}$ does not introduce linear errors in a_i, b_j on each $[c_{i,j}]$. Thus, if we use 2-party programmable tensor product correlations to prepare unauthenticated triples, Step 7 of $\Pi_{\mathsf{patensor}}^k$ for verifying the linear errors is not necessary.*

Instantiation from Unauthenticated Triples. We also note that unauthenticated tensor products can be instantiated from (programmable) unauthenticated Beaver triples in the form of $([a], [b], [c])$ with $c = ab$ [BCG+20]. Therefore, any programmable pseudorandom correlation generator (PCG) for unauthenticated triples can be used for computing the correlation we need. This only requires an additional programmability over the PCGs required for achieving a semi-honest SPDZ protocol with linear communication, which states that the parties' shares of $[a], [b]$ across all the triples are generated from their input seeds via an expansion function, which is a pseudorandom generator (PRG). We give the functionality for the correlation of n-party unauthenticated triples in Figure 6.

Functionality $\mathcal{F}_{\mathsf{uatrip}}^{\mathsf{prog}}(\gamma)$

Let $\mathsf{Expand}^{(\gamma)} : S_\gamma \to \mathbb{F}^\gamma$ be an expansion function with seed space S_γ and output length γ. On receiving $(\mathsf{UATrip}, s_{i,a}, s_{i,b})$ from each party P_i, where $s_{i,a}, s_{i,b} \in S_\gamma$. Then:

1. The trusted party computes each P_i's share of $[\boldsymbol{a}]$ by $\mathsf{Expand}(s_{i,a})$ and P_i's share of $[\boldsymbol{b}]$ by $\mathsf{Expand}(s_{i,b})$, where $\boldsymbol{a} = (a_1, \ldots, a_{m_T})$ and $\boldsymbol{b} = (b_1, \ldots, b_{m_T})$.
2. For each pair of $i = 1, \ldots, m_T$, the trusted party computes $c_i = a_i \cdot b_i$ and receives corrupted parties' shares of $[c_i]$ from Sim. Then, the trusted party samples the honest parties' shares of it based on the secret and the corrupted parties' shares.
3. The trusted party distributes $\{([a_i], [b_i], [c_i])\}_{i=1}^{m_T}$ to the parties.

Abort: If (abort, P) is received from Sim for an honest party P, the trusted party sends abort to P.

Figure 6: Functionality of programmable n-party unauthenticated triples.

For completeness, we provide the protocol $\Pi_{\mathsf{uatensor}}^k$ in the $\mathcal{F}_{\mathsf{uatrip}}^{\mathsf{prog}}(m_U)$-hybrid model for instantiating $\mathcal{F}_{\mathsf{uatensor}}^k$ together with the security proof in Section D.2. The protocol requires $O(k^2)$ invocations of $\mathcal{F}_{\mathsf{uatrip}}^{\mathsf{prog}}(m_U)$ and no additional communication.

4.4 PCG Instantiations

In this section, we briefly discuss the instantiations for the PCGs realizing our required functionalities from various assumptions.

Instantiation from LPN. The most efficient way of realizing $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}$ is to use PCGs from variants of the learning parity with noise (LPN) assumption. A typical instantiation relies on the regular dual-LPN assumption. Informally, the assumption (parameterized by γ, γ', d) states that, given a public matrix H and a random d-regular vector $\boldsymbol{e}$, the vector $H \cdot \boldsymbol{e}$ is computationally indistinguishable from a uniformly random vector. We refer the readers to Section E for the formal definition of the regular dual-LPN assumption. The

expansion function used in $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}$ is defined as follows: Fix a dual-LPN matrix $H \in \mathbb{F}^{\gamma \times \gamma'}$ and consider the seed space $S \subset (\mathbb{F}^{\gamma'})^{m/\gamma}$ containing length-m/γ vectors of d-regular vectors in $\mathbb{F}^{\gamma'}$. The LPN-based expansion function $\mathsf{Expand}_m : S \to \mathbb{F}^m$ is defined by

$$\mathsf{Expand}_m(\boldsymbol{e}_1, \ldots, \boldsymbol{e}_{m/\gamma}) = (H \cdot \boldsymbol{e}_1, \ldots, H \cdot \boldsymbol{e}_{m/\gamma}).$$

$\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}$ is realized using a PCG for tensor product correlations in [BCG+19, BCG+20][2]. The idea of such a PCG is that, to prepare the tensor product for two pseudorandom vectors $\boldsymbol{u}_i, \boldsymbol{x}_i$ from two parties, we can let them be expanded from two sparse vectors $\boldsymbol{e}, \boldsymbol{e}'$ using the expansion function (where $\boldsymbol{e}, \boldsymbol{e}'$ can be chosen by the two parties, providing the programmability). Then, we only need to secret-share $\boldsymbol{e} \otimes \boldsymbol{e}'$, which is also sparse, and then the secret sharing of $\boldsymbol{u}_i \otimes \boldsymbol{x}_i$ can be computed linearly. Due to the progress on function secret sharing (FSS) [GI14, BGI15], $\boldsymbol{e} \otimes \boldsymbol{e}'$ can be secretly shared via a sublinear share size in the vector length of $\boldsymbol{u}_i, \boldsymbol{x}_i$, resulting in a sublinear communication cost in realizing $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}$. In particular, we may choose $\gamma = ck$ for any $c \in \mathbb{Z}^+$ to instantiate $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m, k)$. The instantiation requires an asymptotic seed size of $O(m \cdot \mathsf{poly}(\log \gamma)/\gamma)$ and $O(m\gamma)$ computational cost.

In [RS22], PCG for tensor product correlations is not required for generating the Beaver triples. The parties only need a PCG for n-party vector oblivious linear evaluation (VOLE) and a 2-party programmable PCG for oblivious linear evaluation correlations (OLEs), which replaces each $\boldsymbol{u}_i \otimes \boldsymbol{w}_i$ in $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}$ by $\boldsymbol{u}_i * \boldsymbol{w}_i$, to generate partially authenticated Beaver triples. Notably, we remove the requirement of n-party VOLE correlations, and the programmable PCG for OLEs necessarily relies on the PCG for tensor products if only assuming the regular dual-LPN assumption. In particular, when we choose $\gamma = m$, a single length-m tensor product is generated, and then we extract degree-k tensor products from them. Note that even if we only require Beaver triples from LPN, such an expansion function is the most communication-efficient way for generating m triples. Such a solution does not require the parties to perform any additional communication, meaning that the tensor products come free in communication.

Instantiation from Ring-LPN. The PCG for tensor product correlations is not strictly necessary for generating triples under the Ring-LPN assumption [BCG+20, ARR25]. Nevertheless, the seed-generation procedure in these constructions computes the seeds for the triple correlations using programmable 2-party OLE correlations through a process closely resembling Figure 5. These seeds are themselves derived, via an expansion function, from the seeds determining the parties' shares of $[a]$ and $[b]$ across the triples. Consequently, the PCGs naturally satisfy the programmability. Therefore, we can first use the Ring-LPN-based PCGs to generate unauthenticated triples block by block, and then obtain the tensor product correlations from the unauthenticated triples.

Moreover, the idea of achieving the PCG for 2-party OLEs is similar to that from the regular dual-LPN assumption, and the PCG for OLE from Ring-LPN can naturally be generalized to any degree-2 correlations, as demonstrated in [BCG+20]. More concretely, the Ring-LPN assumption states that given a polynomial a in a ring $R = \mathbb{F}[X]/F(X)$ for a degree-N polynomial $F(X)$, $a \cdot e + e'$ is computationally indistinguishable from a random element in R for random sparse polynomials e, e', i.e. e, e' have random sparse vectors as their coefficients. In the PCG construction for OLEs from Ring-LPN, $\boldsymbol{u}_i$ is expanded from two sparse polynomials (e, e') by computing the coefficients of $u_i = a \cdot e + e'$, and $\boldsymbol{x}_i$ is expanded in the same way from (f, f') with polynomial $x_i = a \cdot f + f'$. Crucially, $u_i \cdot x_i$ can be computed linearly from $(e, e') \otimes (f, f')$. By sharing $(e, e') \otimes (f, f')$ using FSS, a secret sharing of $u_i \cdot x_i$ can be computed, thus providing an OLE over R. When $F(X)$ splits to a product of degree-1 polynomials, the OLE over R is equivalent to N OLEs over $\mathbb{F}$, providing an efficient way for generating OLEs.

Note that in the above approach, the FSS for $(e, e') \otimes (f, f')$ is still instantiated by sharing the tensor products of the coefficient vectors $(\boldsymbol{e}, \boldsymbol{e}'), (\boldsymbol{f}, \boldsymbol{f}')$ of $(e, e'), (f, f')$, and the tensor product $\boldsymbol{u}_i \otimes \boldsymbol{v}_i$ is linear to $(\boldsymbol{e}, \boldsymbol{e}') \otimes (\boldsymbol{f}, \boldsymbol{f}')$. Therefore, the secret sharing of the tensor product $\boldsymbol{u}_i \otimes \boldsymbol{x}_i$ can still be computed in the same way as the LPN-based approach. However, unlike the PCG for OLEs, the computational cost of the PCG from Ring-LPN is similar to that from the regular dual-LPN assumption. We refer the readers to [BCG+20] for more details.

[2]The correlations are called bilinear correlations in [BCG+19, BCG+20].

PCG for Beaver triples can also be prepared from the quasi-abelian syndrome decoding (QA-SD) assumption [BCCD23, LLX+26], which can be regarded as a multivariate version of Ring-LPN. The construction structure is similar to the Ring-LPN-based approach, which can also be generalized to realize our functionalities.

Instantiation from HSS. Another approach for constructing PCGs for unauthenticated triples is based on homomorphic secret sharing (HSS) [BCG+19, CKY25]. Observe that the target correlations can be expressed as additive sharings of $F(r)$ for some function F evaluated on a random input r. The core idea behind the HSS-based approach is to generate the randomness r via a pseudorandom generator PRG, while only secret-sharing the corresponding PRG seed using HSS. As long as the underlying HSS scheme supports evaluating the function $F \circ \mathsf{PRG}(\cdot)$, the desired correlations can be generated efficiently.

Importantly, random unauthenticated triples can be viewed as degree-2 correlations over the parties' shares of $[a], [b]$ across all the triples. Thus, existing HSS-based PCGs already derive the triple correlations from seeds that deterministically expand to the shares of $[a], [b]$, naturally providing the programmability required in our construction. Therefore, we can directly apply the PCGs to generate tensor product correlations.

On the other hand, the main limitation of the HSS-based approach lies in the function class supported by the underlying HSS scheme, which typically requires $F \circ \mathsf{PRG}(\cdot)$ to have low circuit depth. Since the n-party unauthenticated tensor product correlations are also degree-2 correlations, existing HSS-based PCGs for n-party unauthenticated triples can be naturally generalized to generate unauthenticated tensor products as well. A similar observation applies to the approach based on 2-party tensor product correlations: both the 2-party OLE correlations used for generating triples and the 2-party tensor product correlations required for our construction are degree-2 correlations.

4.5 Preprocessing Phase

In this section, we introduce the preprocessing phase of our protocol, which is modeled as a functionality $\mathcal{F}_{\mathsf{prep}}$. The detailed description of $\mathcal{F}_{\mathsf{prep}}$ can be found in Section F. Briefly speaking, $\mathcal{F}_{\mathsf{prep}}$ receives a request $(\mathsf{Prep}, m_R, m_P, m_A, m_T)$ from all the parties and prepares the following sharings for the parties:

- **Random Values:** m_T authenticated sharings in the form of $[\![r]\!]$, where r is a random element in $\mathbb{F}$. These random authenticated sharings are used for masking private wire values and opening them during the evaluation and verification phases.
- **Partially Authenticated Triples:** m_P partially authenticated triples in the form of $([\![a]\!], [\![b]\!], [c])$ with $c = a \cdot b$. These triples are used for letting each party generate authenticated sharings for his inputs.
- **Authenticated Triples:** m_A authenticated triples in the form of $([\![a]\!], [\![b]\!], [\![c]\!])$ with $c = a \cdot b$. These authenticated triples are only used for verification during the online phase.
- **Tensor Products:** m_T length-k partially authenticated tensor products in the form of $\{([\![a_i]\!], [\![b_i]\!])\}_{i=1}^k, \{[a_i \cdot b_j]\}_{i,j=1}^k$ for random elements $\{a_i, b_i\}_{i=1}^k$ in $\mathbb{F}$. The tensor products are used for the inner-product verification.

All the authenticated sharings are corresponding to an additive sharing $[\Delta]$ for a global random MAC key Δ. In our main protocol, we need $m_R = O(m_O + |C| \cdot \log k/k)$ random authenticated sharings, $m_P = m_I + m_O$ partially authenticated triples, $m_A = O(|C|/k)$ authenticated triples, and $m_T = O(|C|/k)$ length-k partially authenticated tensor products, where m_I (resp. m_O) is the number of input (resp. output) wires in the circuit C. We refer the readers to Section 5 and Section 6 for more details about how these preprocessing data are used.

Clearly, $\mathcal{F}_{\mathsf{prep}}$ is easy to realize given $\mathcal{F}_{\mathsf{patensor}}^k$. In particular, random authenticated sharings and partially authenticated triples can be extracted from the tensor products, and the authenticated triples are

computed from partially authenticated triples using the technique of [GLM+24]. For more details about the preprocessing protocol and the security proof, we refer the readers to Section G. We also provide the cost of preprocessing here using the most efficient way of instantiating $\mathcal{F}^k_{\mathsf{patensor}}$ from the two provided approaches. If we need the communication cost of preprocessing to match the semi-honest baseline, we may use PCGs for 2-party tensor products to instantiate $\mathcal{F}^k_{\mathsf{patensor}}$, which requires $O(n^2 \cdot \mathsf{poly}(\log |C|))$ communication and $O(|C|^2 n^2 \cdot \mathsf{poly}(\kappa, \log |C|)$ computation using the LPN-based instantiation [BCG+19]. If we need a lower computational cost, we may choose the second black-box approach from unauthenticated triples, which requires $O(n^2 k^2 \cdot \mathsf{poly}(\log |C|))$ communication and $O(|C| n^2 k \cdot \mathsf{poly}(\kappa, \log |C|))$ computation if the triples are generated from ring-LPN-based PCGs [BCG+20].

5 Evaluation Phase

In this section, we show how the parties share their inputs and perform online evaluation. The online evaluation phase Π_{eval} is run after the preprocessing (i.e., after calling $\mathcal{F}_{\mathsf{prep}}$ for the preprocessing data). The parties first transform the circuit C into a circuit C' of public outputs. Then, they evaluate the circuit C' gate by gate and compute authenticated sharings for:

- All inputs/outputs of C'.
- All input wire values of multiplication gates in C'.

The protocol Π_{eval} is shown in Figure 7.

Protocol Π_{eval}

For an arithmetic circuit C over $\mathbb{F}$ to compute, the parties first transform the circuit to a circuit C' with $C'(x_1 \| \mathsf{mask}_1, \ldots, x_n \| \mathsf{mask}_n) = C(x_1, \ldots, x_n) \oplus (\mathsf{mask}_1, \ldots \mathsf{mask}_n)$. The parties initialize a set $\mathcal{S}_{\mathsf{check}} = \emptyset$ at the beginning. Then:

1. **Authenticating Inputs.** For each input wire of C' attached to party P_i with input value x, the parties consume a partially authenticated triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, [c])$ (from the outputs of $\mathcal{F}_{\mathsf{prep}}$) and send their shares of $[a], [b], [c]$ to P_i. P_i then reconstructs the secrets and verifies whether it holds that $a \cdot b = c$. If not, he aborts the protocol. Otherwise, he sends $x + a$ to all the parties. Finally, the parties obtain $\llbracket x \rrbracket = (x + a) - \llbracket a \rrbracket$.
2. **Evaluating the Circuit.** The multiplication gates of C' are partitioned into $\ell = |C|/k$ groups[a]. The j-th group of multiplication gates has input wire values denoted by $(x_1^{(j)}, y_1^{(j)}), \ldots, (x_k^{(j)}, y_k^{(j)})$ and output wire values denoted by $z_1^{(j)}, \ldots, z_k^{(j)}$. For each gate in C':
   - If the gate is an addition gate with input values x, y and output z, the parties locally compute $[z] = [x] + [y]$.
   - If the gate is a multiplication gate with inputs $x_\alpha^{(j)}, y_\alpha^{(j)}$:
     (a) The parties take the j-th length-k tensor product $\{(\llbracket a_i \rrbracket, \llbracket b_i \rrbracket)\}_{i=1}^k, \{[a_i \cdot b_{i'}]\}_{i,i'=1}^k$ prepared by $\mathcal{F}_{\mathsf{prep}}$ and parse $(\llbracket a_\alpha \rrbracket, \llbracket b_\alpha \rrbracket, [c_{\alpha,\alpha}])$ from it.
     (b) * If $x_\alpha^{(j)}$ (resp. $y_\alpha^{(j)}$) can be linearly computed from inputs of C', then the parties locally compute $\llbracket x_\alpha^{(j)} + a_\alpha \rrbracket = \llbracket x_\alpha^{(j)} \rrbracket + \llbracket a_\alpha \rrbracket$ (resp. $\llbracket y_\alpha^{(j)} + b_\alpha \rrbracket = \llbracket y_\alpha^{(j)} \rrbracket + \llbracket b_\alpha \rrbracket$). Then, the parties run Π_{Open} on $\llbracket x_\alpha^{(j)} + a_\alpha \rrbracket$ (resp. $\llbracket y_\alpha^{(j)} + b_\alpha \rrbracket$).
     * Otherwise, the parties locally compute $[x_\alpha^{(j)} + a_\alpha] = [x_\alpha^{(j)}] + [a_\alpha]$ (resp. $[y_\alpha^{(j)} + b_\alpha] = [y_\alpha^{(j)}] + [b_\alpha]$) and send them to P_{king}. P_{king} reconstructs the secret and sends it back to all the parties. The parties then add $x_\alpha^{(j)}$ (resp. $y_\alpha^{(j)}$) to $\mathcal{S}_{\mathsf{check}}$.
     The parties then locally compute $\llbracket x_\alpha^{(j)} \rrbracket = (x_\alpha^{(j)} + a_\alpha) - \llbracket a_\alpha \rrbracket, \llbracket y_\alpha^{(j)} \rrbracket = (y_\alpha^{(j)} + b_\alpha) - \llbracket b_\alpha \rrbracket$.
     (c) The parties locally compute

$$[z_\alpha^{(j)}] = (x_\alpha^{(j)} + a_\alpha) \cdot (y_\alpha^{(j)} + b_\alpha) - (x_\alpha^{(j)} + a_\alpha) \cdot [b_\alpha] - (y_\alpha^{(j)} + b_\alpha) \cdot [a_\alpha] + [c_{\alpha,\alpha}].$$

   Finally, for each output wire with value v of C', the parties obtain $[v]$. The parties consume a random sharing $\llbracket r \rrbracket$ obtained from the output of $\mathcal{F}_{\mathsf{prep}}$ and do the following:

- If v can be linearly computed from inputs of C', then the parties locally compute $[\![r+v]\!] = [\![r]\!] + [\![v]\!]$. Then, the parties run Π_{Open} on $[\![r+v]\!]$.
- Otherwise, send their shares of $[r+v] = [r] + [v]$ to P_{king}. P_{king} reconstructs $r+v$ and sends it to all the parties. The parties then add v to $\mathcal{S}_{\mathsf{check}}$.

The parties then locally obtain $[\![v]\!] = (r+v) - [\![r]\!]$.

3. **MAC Check.** The parties run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check all the values opened via Π_{Open} during the evaluation phase.

[a] We assume without loss of generality that k divides $|C|$. If not, we only need to add some multiplication gates with public inputs $(0,0)$, such that the multiplication gates can be grouped into $\ell = \lceil |C|/k \rceil$ groups.

Figure 7: Online evaluation protocol.

Since each length-k tensor product facilitates the evaluation of k multiplication gates, the process requires $\lceil |C|/k \rceil$ length-k tensor products. One random authenticated sharing is required for each output wire of C', so the total number of such sharings consumed in Π_{eval} equals the number of output wires in C.

6 Verification Phase

In this section, we show how to verify the correctness of online evaluation. The protocol is provided in Figure 6.

The first step of the verification is to reduce the verification target to an inner-product check. Intuitively, once the circuit inputs and multiplication-gate outputs are fixed, every checked wire is a linear combination of these values. Rather than checking all such circuit relations separately, the parties combine them using a random challenge sampled after the evaluation. Since any errors are fixed before the challenge is chosen, a non-zero error survives this random combination except with negligible probability. By collecting the coefficient of each multiplication term $x_j y_j$, the combined relation becomes a single inner-product relation $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = z$. In particular, the aggregated coefficients are computed after the random challenge is sampled by one reverse circuit traversal through linear gates, which only requires a linear computational cost in $|C|$.

Protocol Π_{ver}

1. **Reducing to Inner-Product Checks.** Denote all pairs of input wire values for the multiplication gates in C' by $(x_1, y_1), \ldots, (x_{|C|}, y_{|C|})$, and denote all the inputs of C' by $v_1, \ldots, v_I$. Let all the values in $\mathcal{S}_{\mathsf{check}}$ be $z_1, \ldots, z_M$, including all the input wire values of multiplication gates and the outputs of C'. Each z_i is computed via a linear combination of $x_1 \cdot y_1, \ldots, x_{|C|} \cdot y_{|C|}$ and $v_1, \ldots, v_I$:

$$\sum_{j=1}^{|C|} \alpha_i^{(j)} x_j y_j + \sum_{j=1}^{I} \beta_i^{(j)} v_j = z_i,$$

where $\{\alpha_i^{(j)}\}_{j=1}^{|C|}, \{\beta_i^{(j)}\}_{j=1}^{I}$ are constants determined by the circuit's linear gates and overall topology for each $i = 1, \ldots, M$. Here, the parties need not explicitly construct the matrix containing all coefficients $\{\alpha_i^{(j)}\}_{j=1}^{|C|}$ and $\{\beta_i^{(j)}\}_{j=1}^{I}$ for $i = 1, \ldots, M$. For each $i = 1, \ldots, M$, $\{\alpha_i^{(j)}\}_{j=1}^{|C|}$ are not all zero (or the checked value can be directly computed from the inputs without evaluating multiplication gates). Then:

(a) The parties send $\mathsf{RandCoin}$ to $\mathcal{F}_{\mathsf{Coin}}$ and obtain a random element $s \in \mathbb{F}^*$.

(b) Recall that the parties have $[\![x_i]\!], [\![y_i]\!]$ for each multiplication gate with input values x_i, y_i, and they also have an authenticated sharing for each input/output of C'. Thus, the parties have the authenticated sharings for all the values in $\mathcal{S}_{\mathsf{check}}$, so they can locally compute

$$[\![\boldsymbol{x}]\!] = \left(\sum_{i=1}^{M} s^i \cdot \alpha_i^{(1)} \cdot [\![x_1]\!], \ldots, \sum_{i=1}^{M} s^i \cdot \alpha_i^{(|C|)} \cdot [\![x_{|C|}]\!] \right), \quad [\![\boldsymbol{y}]\!] = ([\![y_1]\!], \ldots, [\![y_{|C|}]\!])$$

and

$$[\![z]\!] = \sum_{i=1}^{M} s^i [\![z_i]\!] - \sum_{i=1}^{M} \sum_{j=1}^{I} s^i \beta_i^{(j)} [\![v_j]\!].$$

In particular, they compute the aggregated coefficients $\theta_j = \sum_{i=1}^{M} s^i \alpha_i^{(j)}, \eta_j = \sum_{i=1}^{M} s^i \beta_i^{(j)}$ without explicitly constructing the coefficients $\{\alpha_i^{(j)}\}_{j=1}^{|C|}$ and $\{\beta_i^{(j)}\}_{j=1}^{I}$ as follows:
Associate a public accumulator λ_w, initially set to zero, with every wire w of C'. For each $z_i \in \mathcal{S}_{\mathsf{check}}$, update $\lambda_{z_i} \leftarrow \lambda_{z_i} + s^i$. The parties then process the gates of C' in reverse topological order. For each linear gate with output wire value w and input wire values $u_1, \ldots, u_d$ with $w = \sum_{\ell=1}^{d} \gamma_\ell u_\ell$, they update

$$\lambda_{u_\ell} \leftarrow \lambda_{u_\ell} + \gamma_\ell \lambda_w \quad \text{for each } \ell = 1, \ldots, d.$$

When the traversal reaches the output wire of the j-th multiplication gate, whose value is $x_j y_j$, the parties set $\theta_j \leftarrow \lambda_w$ and do not propagate λ_w through this multiplication gate. When it reaches a circuit input v_j, they set $\eta_j \leftarrow \lambda_{v_j}$. The parties then obtain $[\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!], [\![z]\!]$. With high probability, the coefficient $\theta_j = \sum_{i=1}^{M} s^i \alpha_i^{(j)}$ corresponding to $[\![x_j]\!]$ is non-zero for $j = 1, \ldots, |C|$. If not, the parties abort the protocol.

2. **Perform the Inner-Product Check.** The parties run the protocol Π_{IPcheck} (Figure 9) to verify that $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = z$.

Figure 8: Protocol for the verification phase.

Throughout the above process, the aggregated coefficients are computed by one reverse circuit traversal through linear gates. Each linear-gate input edge is processed only once, so the aggregated coefficients are computed in $O(|C|)$ field operations.

The second step is to verify the resulting inner-product relation. The parties first divide it into length-k blocks according to the tensor-product correlations used during evaluation. Within each block, these correlations allow the parties to locally derive all cross-products needed by the verification. The parties then recursively compress each block: in every round, they partition the vectors into λ parts, interpolate them as vector-valued polynomials, and evaluate the polynomials at a fresh random point, reducing the vector length by a factor of λ. An invalid relation remains invalid except with negligible probability. After $\lceil \log_\lambda k \rceil$ rounds, only a scalar multiplication relation remains, which is checked using an authenticated Beaver triple.

The Inner-product check process Π_{IPcheck} is provided below.

Protocol Π_{IPcheck}

To verify that $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = z$ with $[\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!], [\![z]\!]$:

1. **Separate into Blocks.**
   (a) For each group (of index j) of multiplication gates with input wires $(x_1^{(j)}, y_1^{(j)}), \ldots, (x_k^{(j)}, y_k^{(j)})$, let $\boldsymbol{x}^{(j)} = (x_1^{(j)}, \ldots, x_k^{(j)})$ and $\boldsymbol{y}^{(j)} = (y_1^{(j)}, \ldots, y_k^{(j)})$. Let the coefficient corresponding to $[\![x_i^{(j)}]\!]$ in $[\![x]\!]$ be $\theta_i^{(j)}$ and let $\boldsymbol{\theta}^{(j)} = (\theta_1^{(j)}, \ldots, \theta_k^{(j)})$. The parties compute $[z^{(j)}]$ with $z^{(j)} = \langle \boldsymbol{\theta}^{(j)} * \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)} \rangle$ by $[z^{(j)}] = \sum_{i=1}^{k} \theta_i^{(j)} [z_i^{(j)}]$ for each $j = 1, \ldots, \ell = \lceil |C|/k \rceil$, and each $[z_i^{(j)}]$ has been computed during the evaluation phase.
   (b) For each $j = 1, \ldots, \ell - 1$, the parties consume a random sharing $[\![r^{(j)}]\!]$ (from the output of $\mathcal{F}_{\mathsf{prep}}$, same below) and locally compute $[r^{(j)} + z^{(j)}] = [r^{(j)}] + [z^{(j)}]$. Then, they send their shares to P_{king}. P_{king} reconstructs the secret and sends it to all the parties. The parties locally computes $[\![z^{(j)}]\!] = (r^{(j)} + z^{(j)}) - [\![r^{(j)}]\!]$. Finally, the parties compute $[\![z^{(\ell)}]\!] = [\![z]\!] - \sum_{i=1}^{\ell-1} [\![z^{(j)}]\!]$.

   Then, the inner product check is reduced to checking ℓ blocks of inner product checks with vector length k: $\langle \boldsymbol{\theta}^{(j)} * \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)} \rangle = z^{(j)}$ for $j = 1, \ldots, \ell$. In addition, the parties compute $[(\boldsymbol{\theta}^{(j)} * \boldsymbol{x}^{(j)}) \otimes \boldsymbol{y}^{(j)}]$ by

$$\begin{aligned}[\theta_\alpha^{(j)} x_\alpha^{(j)} \cdot y_\beta^{(j)}] =& \theta_\alpha^{(j)} \cdot (x_\alpha^{(j)} + a_\alpha) \cdot (y_\beta^{(j)} + b_\beta) - \theta_\alpha^{(j)} \cdot (x_\alpha^{(j)} + a_\alpha) \cdot [b_\beta] \\ & - \theta_\alpha^{(j)} \cdot (y_\beta^{(j)} + b_\beta) \cdot [a_\alpha] + \theta_\alpha^{(j)} \cdot [c_{\alpha,\beta}]\end{aligned}$$

for each pair of $(\alpha, \beta) \in \{1, \ldots, k\}^2$, where $\{[\![a_\alpha]\!], [\![b_\alpha]\!]\}_{\alpha=1}^k, \{[c_{\alpha,\beta}]\}_{\alpha,\beta=1}^k$ is the length-k tensor product used for evaluating the j-th group of multiplication gates during the evaluation phase.

2. **Recursive Check.** To check each inner product $\langle \boldsymbol{\theta}^{(j)} * \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)} \rangle = z^{(j)}$, we use a recursive reduction on vector length. The parties initialize $[\![\boldsymbol{u}]\!] = \boldsymbol{\theta}^{(j)} * [\![\boldsymbol{x}^{(j)}]\!], [\![\boldsymbol{v}]\!] = [\![\boldsymbol{y}^{(j)}]\!], [\![w]\!] = [\![z^{(j)}]\!]$, and let $[\boldsymbol{u} \otimes \boldsymbol{v}] = [(\boldsymbol{\theta}^{(j)} * \boldsymbol{x}^{(j)}) \otimes \boldsymbol{y}^{(j)}]$. Then, the parties repeat the following until the length of $\boldsymbol{u}, \boldsymbol{v}$ is reduced to 1:
   (a) The parties partition $\boldsymbol{u}$ into λ vectors $(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_\lambda)$, where each $\boldsymbol{u}_i$ has the same length ℓ' (if λ is not a factor of the vector length, the parties add zeros after $\boldsymbol{u}$). Similarly, $\boldsymbol{v}$ is partitioned into $(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\lambda)$.
   (b) The parties locally compute $[w_i]$ with $w_i = \langle \boldsymbol{u}_i, \boldsymbol{v}_i \rangle$ from $[\boldsymbol{u} \otimes \boldsymbol{v}]$.
   (c) For each $i = 1, \ldots, \lambda - 1$, the parties consume a random sharing $[\![r_i]\!]$ and locally compute $[r_i + w_i] = [r_i] + [w_i]$. Then, they send their shares to P_{king}. P_{king} reconstructs the secret and sends it to all the parties. The parties locally compute $[\![w_i]\!] = (r_i + w_i) - [\![r_i]\!]$. Finally, the parties compute $[\![w_\lambda]\!] = [\![w]\!] - \sum_{i=1}^{\lambda-1} [\![w_i]\!]$.
   (d) Each party P_j computes a vector of ℓ degree-$(\lambda - 1)$ polynomials $\boldsymbol{f}_j(\cdot)$ such that $\boldsymbol{f}_j(i)$ equals his share of $[\boldsymbol{u}_i]$ for each $i = 1, \ldots, \lambda$. Similarly, P_j computes a vector of degree-$(\lambda - 1)$ polynomials $\boldsymbol{g}_j(\cdot)$ such that $\boldsymbol{g}_j(i)$ equals his share of $[\boldsymbol{v}_i]$ for each $i = 1, \ldots, \lambda$. Let $\boldsymbol{f}(\cdot) = \boldsymbol{f}_1(\cdot) + \cdots + \boldsymbol{f}_n(\cdot)$ and $\boldsymbol{g}(\cdot) = \boldsymbol{g}_1(\cdot) + \cdots + \boldsymbol{g}_n(\cdot)$.
   (e) Let $h(\cdot) = \langle \boldsymbol{f}(\cdot), \boldsymbol{g}(\cdot) \rangle$, which is a degree-$(2\lambda - 2)$ polynomial. Then, we need to verify that $w_i = h(i)$ for $i = 1, \ldots, \lambda$. For each $i = \lambda + 1, \ldots, 2\lambda - 1$, let $\boldsymbol{c}_i = (c_i^{(1)}, \ldots, c_i^{(\lambda)})$ be the coefficient vector such that $\boldsymbol{f}(i) = \sum_{j=1}^{\lambda} c_i^{(j)} \boldsymbol{u}_j$. The parties then locally compute
   $$[h(i)] = \sum_{\alpha=1}^{\lambda} \sum_{\beta=1}^{\lambda} c_i^{(\alpha)} c_i^{(\beta)} [\langle \boldsymbol{u}_\alpha, \boldsymbol{v}_\beta \rangle],$$
   where each $[\langle \boldsymbol{u}_\alpha, \boldsymbol{v}_\beta \rangle]$ is computed locally from $[\boldsymbol{u} \otimes \boldsymbol{v}]$.
   (f) For each $i = \lambda + 1, \ldots, 2\lambda - 1$, the parties consume a random sharing $[\![r_i]\!]$ and locally compute $[r_i + h(i)] = [r_i] + [h(i)]$. Then, they send their shares to P_{king}. P_{king} reconstructs the secret and sends it to all the parties. The parties locally compute $[\![h(i)]\!] = (r_i + h(i)) - [\![r_i]\!]$.
   (g) The parties send $\mathsf{RandCoin}$ to $\mathcal{F}_{\mathsf{Coin}}$ and receive $r \in \mathbb{F}$[a]. Since $\boldsymbol{f}(\cdot), \boldsymbol{g}(\cdot)$ are vectors of degree-$(\lambda - 1)$ polynomials, $\boldsymbol{f}(r), \boldsymbol{g}(r)$ can be computed from $(\boldsymbol{f}(1), \ldots, \boldsymbol{f}(\lambda)), (\boldsymbol{g}(1), \ldots, \boldsymbol{g}(\lambda))$ respectively via a linear function L. The parties then locally compute $[\![\boldsymbol{f}(r)]\!] = L([\![\boldsymbol{f}(1)]\!], \ldots, [\![\boldsymbol{f}(\lambda)]\!])$ and $[\![\boldsymbol{g}(r)]\!] = L([\![\boldsymbol{g}(1)]\!], \ldots, [\![\boldsymbol{g}(\lambda)]\!])$. Similarly, since $h(\cdot)$ is of degree $(2\lambda - 2)$, there exists a linear function L' such that $h(x) = L'(h(1), \ldots, h(2\lambda - 1))$. The parties then locally compute $[\![h(r)]\!] = L'([\![w_1]\!], \ldots, [\![w_\lambda]\!], [\![h(\lambda + 1)]\!], \ldots, [\![h(2\lambda - 1)]\!])$.
   (h) Assume $L(\boldsymbol{s}) = \langle \boldsymbol{c}, \boldsymbol{s} \rangle$. The parties update $[\boldsymbol{u} \otimes \boldsymbol{v}]$ by
   $$[\boldsymbol{u} \otimes \boldsymbol{v}] = \langle \boldsymbol{c} \otimes \boldsymbol{c}, [(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_\lambda) \otimes (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\lambda)] \rangle,$$
   where $(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_\lambda) \otimes (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\lambda)$ means the length-λ^2 vector in $(\mathbb{F}^{\ell'^2})^{\lambda^2}$ containing $\boldsymbol{u}_\alpha \otimes \boldsymbol{v}_\beta$ for each pair of $\alpha, \beta \in \{1, \ldots, \lambda\}$. In particular, $[(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_\lambda) \otimes (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\lambda)]$ can be extracted from $[\boldsymbol{u} \otimes \boldsymbol{v}]$. Finally, the parties update $[\![\boldsymbol{u}]\!] = [\![\boldsymbol{f}(r)]\!], [\![\boldsymbol{v}]\!] = [\![\boldsymbol{g}(r)]\!]$, and $[\![w]\!] = [\![h(r)]\!]$.

3. **Verify the Final Triple.** After Step 2, the parties finally obtain a single triple $([\![u]\!], [\![v]\!], [\![w]\!])$ for each group of multiplication gates, where we need to verify that $w = u \cdot v$. Then:
   (a) The parties consume an authenticated triple $([\![a]\!], [\![b]\!], [\![c]\!])$ obtained from $\mathcal{F}_{\mathsf{prep}}$, locally compute $[\![u + a]\!], [\![v + b]\!]$, and run Π_{Open} on them. Then, the parties locally compute
   $$[\![\tau]\!] = (u + a) \cdot (v + b) - (u + a) \cdot [\![b]\!] - (v + b) \cdot [\![a]\!] + [\![c]\!] - [\![w]\!].$$
   (b) The parties run Π_{Open} on $[\![\tau]\!]$ and check whether $\tau = 0$. If not, they abort the protocol.

   Finally, the parties run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check all the opened values during the check, i.e., the values $u + a, v + b, \tau$ for all ℓ groups of multiplication gates.

[a] This random coin can be used to check all the blocks of inner products.

Figure 9: Verification for the inner-product correlation.

During verification, separating the vectors into blocks requires $\ell - 1$ random authenticated sharings. In addition, $2(\lambda-1)$ random sharings are consumed during each reduction in vector length. Since the reduction needs to be performed $\ell \cdot \lceil \log_\lambda k \rceil$ times, $2\ell(\lambda-1) \cdot \lceil \log_\lambda k \rceil$ random sharings are required in total. For the check of final triples, $\ell = \lceil |C|/k \rceil$ authenticated triples are required. The computational cost of this step is $O(|C|nk)$, as the multiplication gates are divided into $O(|C|/k)$ blocks, and the recursive check of each block requires $O(k^2)$ local operations.

7 Main Protocol

Finally, we put everything together in our main protocol Π_{main}. The protocol runs the preprocessing, online, and verification phases successively. As we have counted, in total, we require $m_R = m_O + 2\ell(\lambda-1) \cdot \lceil \log_\lambda k \rceil + \ell - 1$ random authenticated sharings (where $\ell = \lceil |C|/k \rceil$), $m_P = m_I + m_O$ partially authenticated triples, $m_A = \ell$ authenticated triples, and $m_T = \lceil |C|/k \rceil$ partially authenticated length-k tensor products in the preprocessing. After the verification, the parties run an output reconstruction protocol, where the parties open the secrets of outputs and finally call $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check their correctness. We provide the detailed description of Π_{main} together with the security proof in Section H.

Regardless of instantiating the PCGs, the total communication cost is $4|C|n + 4m_I n + 8m_O n + O(|C|n\lambda k^{-1} \cdot \log_\lambda k) + n^2 \mathsf{poly}(\kappa)$, where $|C|, m_I, m_O$ are the numbers of multiplication gates, input wires, and output wires in C, respectively. This results in an overall communication cost of $4|C|n + 4m_I n + 8m_O n + o(|C|)$ by taking $\lambda = 2$ and $k = \omega(n \log n)$, giving an amortized communication cost of $4n$ per multiplication gate. The total computational cost is $O(|C|nk)$ beyond the PCGs. We refer the readers to Section I for a detailed cost analysis, including the analysis for PCGs.

8 Further Reducing Communication Cost via Turbospeedz

In this section, we show how to further reduce the amortized communication cost of our protocol to $2n$ per multiplication gate using Turbospeedz online evaluation [BNO19]. The protocol Π_{turbo} achieving this target requires the preprocessing of tensor product correlations for length-$O(|C|)$ vectors and computing, which incurs a quadratic computational complexity in $|C|$. Furthermore, the online computational cost is also quadratic in $|C|$. This provides a trade-off between communication and computational complexities.

Preprocessing. In particular, the preprocessing is similar to that of our main protocol. The only difference is that the length-k tensor products and the input mask sharings are replaced by the preprocessing of $[\![\boldsymbol{w}]\!], [\boldsymbol{w} \otimes \boldsymbol{w}]$ for a random vector $\boldsymbol{w} = (w_1, \ldots, w_m)$ with $m = 2m_I + 2m_O + |C|$, where we denote the sharing $[w_i \cdot w_j]$ from $[\boldsymbol{w} \otimes \boldsymbol{w}]$ by $[w_{i,j}]$ for each pair of $(i,j) \in \{1, \ldots, m_C\}^2$. In particular, $[\![w_i]\!]$ for $i = 1, \ldots, m_I + m_O$ serves as the input mask sharings, and $[\![w_{m_I+m_O+j}]\!]$ for $j = 1, \ldots, |C|$ serve as the mask sharings for the $|C|$ output wires of multiplication gates in the circuit. For each $i = 1, \ldots, m_I + m_O$, $([\![w_i]\!], [\![w_{|C|+m_I+m_O+i}]\!], [w_{i,|C|+m_I+m_O+i}])$ ser serves as a partially authenticated triple that helps a party generate an authenticated sharing for an input to C'. The preprocessing functionality $\mathcal{F}_{\mathsf{Turboprep}}$ is provided in Section J.

The instantiation of $\mathcal{F}_{\mathsf{Turboprep}}$ is similar to that of $\mathcal{F}_{\mathsf{prep}}$. The additional task of preparing $\{[\![w_i]\!]\}_{i=1}^m$, $\{[w_{i,j}]\}_{i,j=1}^m$ can be regarded as a single length-m partially authenticated tensor product with $a_i = b_i = w_i$ for each $i = 1, \ldots, m$. The parties only need to follow the approach in Section 4 to prepare it, except that $\{[\![b_i]\!]\}_{i=1}^k$ is not prepared in parallel with $\{[\![a_i]\!]\}_{i=1}^k$ (namely $\{[\![w_i]\!]\}_{i=1}^m$ in $\mathcal{F}_{\mathsf{Turboprep}}$). Instead, $\{[\![b_i]\!]\}_{i=1}^k$ are directly set to $\{[\![a_i]\!]\}_{i=1}^k$. Note that during both protocols we introduced in Section 4.3 to instantiate $\mathcal{F}_{\mathsf{patensor}}^k$, the shares of $\{[\![a_i]\!], [\![b_i]\!]\}_{i=1}^k$ are programmed from chosen seeds. Therefore, we only need to let the parties use the same seed to generate their shares of $\{[\![a_i]\!]\}_{i=1}^k$ and $\{[\![b_i]\!]\}_{i=1}^k$, while all other steps remain the same.

Evaluation. The online evaluation follows the framework of [BNO19]. More concretely, each output of a multiplication gate is computed directly from all the outputs of multiplication gates and the input wires of the circuit via a degree-2 sub-circuit, using $\{[\![w_i]\!]\}_{i=1}^m$, $\{[w_{i,j}]\}_{i,j=1}^m$ prepared by $\mathcal{F}_{\mathsf{Turboprep}}$.

Protocol $\Pi_{\mathsf{turboeval}}$

For an arithmetic circuit C over $\mathbb{F}$ to compute, the parties first transform the circuit to a circuit C' with $C'(x_1\|\mathsf{mask}_1,\ldots,x_m\|\mathsf{mask}_n) = C(x_1,\ldots,x_n)\oplus(\mathsf{mask}_1,\ldots\mathsf{mask}_n)$. The input wires of C' attached to $P_1,\ldots,P_n$ are indexed in order by $1,\ldots,m_I+m_O$. The output wires of multiplication gates are then indexed in topological order by $m_1+m_O+1,\ldots,m_I+m_O+|C|$. Then:

1. **Authenticating Inputs.** For each input wire of index j of C' attached to party P_i with input value z_j, the parties send their shares of $[w_j],[w_{m_I+m_O+|C|+j}],[w_{j,m_I+m_O+|C|+j}]$ to P_i. P_i then reconstructs the secrets and verifies whether $w_j\cdot w_{m_I+m_O+|C|+j} = w_{j,m_I+m_O+|C|+j}$. If not, he aborts the protocol. Otherwise, he sends z_j+w_j to all the parties. Finally, the parties obtain $[\![z_j]\!] = (z_j+w_j)-[\![w_j]\!]$.
2. **Evaluating the Circuit.** For $i = m_I+m_O+1$ to $m_I+m_O+|C|$:
   (a) The value z_j+w_j has been publicly reconstructed for each $j=1,\ldots,i-1$. For each $\alpha=1,\ldots,i-2$, the parties locally compute $[z_\alpha\cdot z_{i-1}]$ by
   $$(z_\alpha+w_\alpha)\cdot(z_{i-1}+w_{i-1})-(z_\alpha+w_\alpha)\cdot[w_{i-1}]-(z_{i-1}+w_{i-1})\cdot[w_\alpha]+[w_{\alpha,i-1}].$$
   (b) Let x_i,y_i be the input values for the multiplication gate whose output wire has index i, and let z_i be its output wire value. Then, x_i can be written as $\sum_{j=1}^{i-1}\beta_j\cdot z_j$ for some coefficients $\beta_1,\ldots,\beta_{i-1}\in\mathbb{F}$ defined by the circuit C'. y_i can be written similarly as $\sum_{j=1}^{i-1}\gamma_j\cdot z_j$. Then, the parties locally compute
   $$[z_i] = \sum_{j_1=1}^{i-1}\sum_{j_2=1}^{i-1}\beta_{j_1}\gamma_{j_2}[z_{j_1}\cdot z_{j_2}].$$
   (c) The parties locally compute their shares of $[z_i+w_i]$ and send them to P_{king}. P_{king} then reconstructs the secret z_i+w_i and sends it to all the parties. The parties then locally compute $[\![z_i]\!] = (z_i+w_i)-[\![w_i]\!]$.

   After evaluating all the multiplication gates, the parties obtain $[\![z_1]\!],\ldots,[\![z_m]\!]$ for $m = m_I+m_O+|C|$. Since all the wire values in C' can be written as linear combinations of $z_1,\ldots,z_m$, the parties can locally compute the authenticated sharing for all of them.

Figure 10: Online evaluation protocol of Π_{turbo}.

Verification. After the evaluation, the parties have obtained an authenticated sharing for all the wire values in C', together with public values z_i+w_i for each $i=1,\ldots,m$. Since outputs of multiplication gates are authenticated, we only need to verify the correctness of multiplications ($\mathcal{S}_{\mathsf{check}}$ contains all outputs for multiplication gates in C').

In particular, for each multiplication gate with inputs $x_i=\sum_{j=1}^{i-1}\beta_j\cdot z_j, y_i=\sum_{j=1}^{i-1}\gamma_j\cdot z_j$, we can regard that the parties as using a triple $[\![a_i]\!]=\sum_{j=1}^{i-1}\beta_j\cdot[\![w_j]\!], [\![b_i]\!]=\sum_{j=1}^{i-1}\gamma_j\cdot[\![w_j]\!], [c_i]=\sum_{j_1=1}^{i-1}\sum_{j_2=1}^{i-1}\beta_{j_1}\gamma_{j_2}[w_{j_1,j_2}]$ to evaluate the multiplication gate. Then, the parties have obtained x_i+a_i, y_i+b_i for each multiplication gate with output z_i, which is the same as the evaluation phase Π_{eval} for our main protocol Π_{main}. Furthermore, each $[a_i\cdot b_j]$ is still a linear combination of $\{[w_{\alpha,\beta}]\}_{\alpha,\beta=1}^m$, so they can be computed by the parties locally. In other words, we can regard the multiplication gates as being evaluated using a single length-$|C|$ tensor product. Therefore, the correctness of the evaluation can be verified in the same way as in Π_{ver}. We refer the reader to Section K for a detailed protocol description Π_{turbover} for the verification phase of Π_{turbo}.

Protocol Description. In the final protocol Π_{turbo}, the parties call $\mathcal{F}_{\mathsf{Turboprep}}$, run the evaluation phase $\Pi_{\mathsf{turboeval}}$ and the verification phase Π_{turbover}, and run a similar output reconstruction step as in Π_{main}. We refer the readers to Section L for the detailed description of Π_{turbo}.

The security proof for Π_{turbo} is similar to that of Π_{main}. We refer the readers to Section M for a proof sketch. Clearly, except for the input/output steps (where $2m_O n$ communication cost is saved for authenticating outputs) and the evaluation step in $\Pi_{\mathsf{turboeval}}$, the communication cost of all other steps is equal to the cost of Π_{main} in the case $k = |C|$, which is $O(n\lambda \log_\lambda |C|) + n^2\mathsf{poly}(\kappa)$. By taking $\lambda = 2$, the additional per-gate communication overhead is $O(n \log |C|/|C| + n^2\mathsf{poly}(\kappa)/|C|) = o(1)$ for large enough circuits. Since the parties only need to authenticate the output wire values of multiplication gates, we save an authentication process per multiplication gate compared to Π_{main}, resulting in an amortized communication of $2n$ per multiplication gate.

References

[ARR25] Amit Agarwal, Srinivasan Raghuraman, and Peter Rindal. Fully distributed multi-point functions for PCGs and beyond. Cryptology ePrint Archive, Report 2025/2294, 2025.

[AS22] Damiano Abram and Peter Scholl. Low-communication multiparty triple generation for SPDZ from ring-LPN. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *PKC 2022: 25th International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 13177 of *Lecture Notes in Computer Science*, pages 221–251, Virtual Event, March 8–11, 2022. Springer, Cham, Switzerland.

[BCCD23] Maxime Bombar, Geoffroy Couteau, Alain Couvreur, and Clément Ducros. Correlated pseudorandomness from the hardness of quasi-abelian decoding. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 567–601, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Cham, Switzerland.

[BCG+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 489–518, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Cham, Switzerland.

[BCG+20] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-LPN. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 387–416, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Cham, Switzerland.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 896–912, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[BCS19] Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using TopGear in overdrive: A more efficient ZKPoK for SPDZ. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019: 26th Annual International Workshop on Selected Areas in Cryptography*, volume 11959 of *Lecture Notes in Computer Science*, pages 274–302, Waterloo, ON, Canada, August 12–16, 2019. Springer, Cham, Switzerland.

[BGH+25] Elette Boyle, Niv Gilboa, Matan Hamilis, Yuval Ishai, and Ariel Nof. Preprocessing for life: Dishonest-majority MPC with a trusted or untrusted dealer. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *2025 IEEE Symposium on Security and Privacy*, pages 2433–2452, San Francisco, CA, USA, May 12–15, 2025. IEEE Computer Society Press.

[BGH+26] Elette Boyle, Niv Gilboa, Matan Hamilis, Yuval Ishai, and Ariel Nof. Dishonest-majority secure computation via PIR-authenticated multiplication triples. In *CRYPTO 2026*, 2026.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 337–367, Sofia, Bulgaria, April 26–30, 2015. Springer Berlin Heidelberg, Germany.

[BGIN20] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Efficient fully secure computation via distributed zero-knowledge proofs. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 244–276, Daejeon, South Korea, December 7–11, 2020. Springer, Cham, Switzerland.

[BGIN21] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Sublinear GMW-style compiler for MPC with preprocessing. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 457–485, Virtual Event, August 16–20, 2021. Springer, Cham, Switzerland.

[BGIN22] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Secure multiparty computation with sublinear preprocessing. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 427–457, Trondheim, Norway, May 30 – June 3, 2022. Springer, Cham, Switzerland.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[BNO19] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online SPDZ! Improving SPDZ using function dependent preprocessing. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 2019: 17th International Conference on Applied Cryptography and Network Security*, volume 11464 of *Lecture Notes in Computer Science*, pages 530–549, Bogota, Colombia, June 5–7, 2019. Springer, Cham, Switzerland.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19. ACM, 1988.

[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 Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland.

[CKY25] Geoffroy Couteau, Naman Kumar, and Xiaxi Ye. Multiparty homomorphic secret sharing and more from LPN and MQ. In Benny Applebaum and Huijia (Rachel) Lin, editors, *TCC 2025: 23rd Theory of Cryptography Conference, Part II*, volume 16269 of *Lecture Notes in Computer Science*, pages 498–530, Aarhus, Denmark, December 1–5, 2025. Springer, Cham, Switzerland.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, September 9–13, 2013. Springer Berlin Heidelberg, Germany.

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

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany.

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

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 640–658, Copenhagen, Denmark, May 11–15, 2014. Springer Berlin Heidelberg, Germany.

[GLM+24] Vipul Goyal, Junru Li, Ankit Kumar Misra, Rafail Ostrovsky, Yifan Song, and Chenkai Weng. Dishonest majority constant-round MPC with linear communication from DDH. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024, Part VI*, volume 15489 of *Lecture Notes in Computer Science*, pages 167–199, Kolkata, India, December 9–13, 2024. Springer, Singapore, Singapore.

[GLO+21] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. ATLAS: Efficient and scalable MPC in the honest majority setting. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 244–274, Virtual Event, August 16–20, 2021. Springer, Cham, Switzerland.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, 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.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 830–842, Vienna, Austria, October 24–28, 2016. ACM Press.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Cham, Switzerland.

[LLX+26] Zhe Li, Hongqing Liu, Chaoping Xing, Yizhou Yao, and Chen Yuan. Faster pseudorandom correlation generators via walsh-hadamard transform. Cryptology ePrint Archive, Paper 2026/196, 2026.

[LN17] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 259–276, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[LOS14] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 495–512, Santa Barbara, CA, USA, August 17–21, 2014. Springer Berlin Heidelberg, Germany.

[LXYY26] Zhe Li, Chaoping Xing, Yizhou Yao, and Chen Yuan. On the power of sumcheck in secure multiparty computation. PKC 2026, 2026.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In David S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA*, pages 73–85. ACM, 1989.

[RS22] Rahul Rachuri and Peter Scholl. Le mans: Dynamic and fluid MPC for dishonest majority. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part I*, volume 13507 of *Lecture Notes in Computer Science*, pages 719–749, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Cham, Switzerland.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.

[Yao82] Andrew Chi-Chih Yao. Theory and applications of trapdoor functions (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 80–91. IEEE Computer Society, 1982.

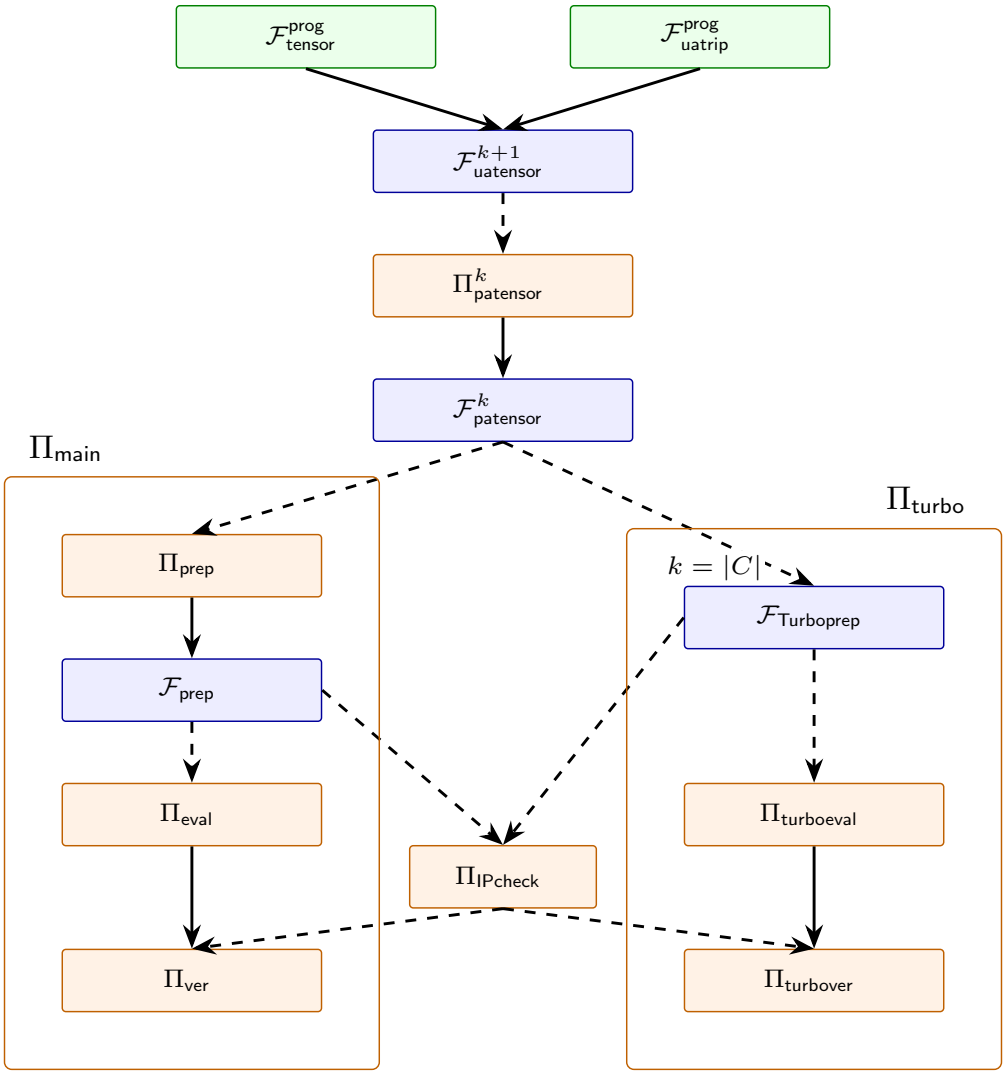


Figure 11: Overview of the functionalities and protocols used in our construction.

A Standard Subprotocols

We give the standard functionalities for subprotocols of generating commitments and random public coins in this section. The functionality $\mathcal{F}_{\mathsf{Commit}}$ for commitments is given in Figure 12.

Functionality $\mathcal{F}_{\mathsf{Commit}}$

Commit: On input $(\mathsf{commit}, P_i, x, \mathsf{mid}_x)$ from P_i, where mid_x is the message index of x, the trusted party stores (P_i, x, mid_x) and sends $(\mathsf{committed}, P_i, \mathsf{mid}_x)$ to all the parties. In particular, if the commitment request has been called for the same message index before, the trusted party ignores the request.

Open: On input $(\mathsf{open}, P_i, \mathsf{mid}_x)$ from all the honest parties, the trusted party retrieves x and sends (x, mid_x) to all the parties.

Figure 12: Functionality for commitment.

The coin generation functionality $\mathcal{F}_{\mathsf{Coin}}$ is given in Figure 13. To realize $\mathcal{F}_{\mathsf{Coin}}$, each party commits a

randomly chosen element in $\mathbb{F}^*$ and then opens it to all parties. Then, the product of the chosen values is taken to be the random coin. Finally, the parties can perform a cross-check to make sure that the received coins are the same.

Functionality $\mathcal{F}_{\mathsf{Coin}}$

1. On receiving $\mathsf{RandCoin}$ from all the honest parties, the trusted party samples $r \in \mathbb{F}^*$.
2. The trusted party sends r to Sim. If abort is received from Sim, the trusted party sends abort to all the honest parties and aborts the functionality. Otherwise, the trusted party sends r to all the parties.

Figure 13: Functionality for generating a common coin.

B MAC Check on Opened Values

With random coins and commitments, the opened authenticated values are able to be checked. In particular, for an authenticated sharing $[\![x]\!]$ whose secret x needs to be reconstructed to all the parties, the parties call $\Pi_{\mathsf{Open}}([\![x]\!])$ (see Figure 14), where P_{king} can be any of the parties.

Protocol $\Pi_{\mathsf{Open}}([\![x]\!])$

1. All the parties send their shares of $[x]$ to P_{king}.
2. P_{king} reconstructs x and sends it to all the parties.

Figure 14: Protocol to open a SPDZ sharing.

When the secrets of some authenticated sharings are opened via Π_{Open}, they can check their correctness via a standard SPDZ MAC-check protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ [DKL+13] in the $\{\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model (see Figure 15).

Protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$

Let all the parties agree on a pseudorandom generator $\mathsf{PRG} : \mathbb{F} \to \mathbb{F}^m$. The parties want to check the MACs on opened values $(A_1, \ldots, A_m)$ for sharings $([\![A_1]\!], \ldots, [\![A_m]\!])$.

1. The parties call $\mathcal{F}_{\mathsf{Coin}}$ to get a random element χ in $\mathbb{F}^*$.
2. The parties compute $A = \sum_{i=1}^{m} \chi^i \cdot A_i$ and $[\gamma] = \sum_{i=1}^{m} \chi^i \cdot [\Delta \cdot A_i]$.
3. The parties compute $[\sigma] = [\gamma] - [\Delta] \cdot A$. Each P_i calls $\mathcal{F}_{\mathsf{Commit}}$ with input $(\mathsf{commit}, P_i, \sigma_i, \mathsf{mid}_{\sigma_i})$, where σ_i is P_i's share of $[\sigma]$.
4. The parties open their commitments via sending $(\mathsf{open}, P_i, \mathsf{mid}_{\sigma_i})$ for each party P_i and check that $\sum_{i=1}^{n} \sigma_i = 0$. If not, the parties abort the protocol.

Figure 15: Protocol for MAC checking.

C Security Proof for the Preparation of Partially Authenticated Tensor Products

We prove that the protocol $\Pi^k_{\mathsf{patensor}}$ (Figure 3) securely computes $\mathcal{F}^k_{\mathsf{patensor}}$ in the $\{\mathcal{F}^{k+1}_{\mathsf{uatensor}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model here. We prove the security by constructing an ideal adversary Sim. Then we will show that the output in the ideal world is indistinguishable from that in the real world against the adversary $\mathcal{A}$ using hybrid arguments. Since the parties have no inputs to $\mathcal{F}^k_{\mathsf{patensor}}$, and the outputs of corrupted parties are fully determined by Sim in the ideal world (so there are no restrictions on honest parties' outputs other than one honest party's), we can simply consider the maximum corruption case, where there is only one honest party.

Without loss of generality, we assume that P_{king} is corrupted and let P_h be the honest party. If another party is honest, Sim just needs to honestly emulate him to run the protocol.

Simulator Sim

1. Sim initializes ErrorCheck = MACCheck = 0.
2. Sim emulates $\mathcal{F}_{\mathsf{uatensor}}^{k+1}$ to receive $(\mathsf{UATens}, m_T + \lceil m_T/k \rceil + 1)$ from corrupted parties and receive corrupted parties' outputs together with the coefficients for linear errors from $\mathcal{A}$. Then, Sim emulates $\mathcal{F}_{\mathsf{uatensor}}^{k+1}$ to distribute the corrupted parties' outputs to them. If $\delta_{a,i,j}^{(\alpha)} \neq 0$ or $\delta_{b,i,j}^{(\alpha)} \neq 0$ for any $i, j \in \{1, \ldots, k\}$ and $\alpha \in \{1, \ldots, m_T + \lceil m_T/k \rceil + 1\}$, Sim sets ErrorCheck = 1.
3. Sim samples two random field elements as P_h's shares of $[\Delta + \Delta_a^{(\alpha)}], [\Delta + \Delta_b^{(\alpha)}]$ for each $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$. Sim then sends them to P_{king} on behalf of P_h.
4. For each $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$, Sim receives $\overline{\Delta + \Delta_a^{(\alpha)}}, \overline{\Delta + \Delta_b^{(\alpha)}}$ from P_{king}. Sim computes the sum of corrupted parties' shares of $[\Delta]$ from the sum of their shares of $[\Delta_a^{(1)}]$ and $[\Delta + \Delta_a^{(1)}]$, where the sum of their shares of $[\Delta + \Delta_a^{(1)}]$ are computed assuming that P_{king} honestly computes $\Delta + \Delta_a^{(1)} = \overline{\Delta + \Delta_a^{(1)}}$. Sim then reconstructs $\Delta + \Delta_a^{(\alpha)}, \Delta + \Delta_b^{(\alpha)}$ from all parties' shares for each $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$.
5. Sim computes $\delta_a^{(\alpha)} = \overline{\Delta + \Delta_a^{(\alpha)}} - (\Delta + \Delta_a^{(\alpha)})$ and $\delta_b^{(\alpha)} = \overline{\Delta + \Delta_b^{(\alpha)}} - (\Delta + \Delta_b^{(\alpha)})$. Unless $\nu = \delta_a^{(\alpha)} - \delta_{a,i}^{(\alpha)} = \delta_b^{(\alpha)} - \delta_{b,i}^{(\alpha)}$ and the value are all the same for $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil$ and $i = 1, \ldots, k$, Sim sets MACCheck = 1. Otherwise, Sim adds ν into the sum of corrupted parties' shares of $[\Delta]$, sets P_{king}'s share to be this sum, and sets all other corrupted parties' shares of $[\Delta]$ to 0.
6. For each $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$, Sim follows the protocol to compute corrupted parties' shares of $[\Delta \cdot a_i^{(\alpha)}] = (\Delta + \Delta_b^{(\alpha)}) \cdot [a_i^{(\alpha)}] - [e_i^{(\alpha)}], [\Delta \cdot b_i^{(\alpha)}] = (\Delta + \Delta_a^{(\alpha)}) \cdot [b_i^{(\alpha)}] - [d_i^{(\alpha)}]$ for each $i = 1, \ldots, k$.
7. Sim receives RandCoin from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send r to them. If Sim receives abort from $\mathcal{A}$, he sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view. Sim follows the protocol to compute
$$\mathsf{PRG}(r) = (\{r_{a,1}^{(\alpha)}, \ldots, r_{a,k}^{(\alpha)}, r_{b,1}^{(\alpha)}, \ldots, r_{b,k}^{(\alpha)}\}_{\alpha=1}^{m_T+\lceil m_T/k \rceil+1}).$$
If for all $i = 1, \ldots, k$, $r_{a,i}^{(m_T+\lceil m_T/k \rceil+1)} = r_{b,i}^{(m_T+\lceil m_T/k \rceil+1)} = 0$, Sim aborts the simulation. Then, Sim follows the protocol to compute corrupted parties' shares of $[\![A]\!]$.
8. Sim samples a random field element as P_h's share of $[A]$ and sends it to P_{king} on behalf of P_h. Then, Sim reconstructs A from all parties' shares of $[A]$. Sim then receives $\overline{A}$ from P_{king} and computes $\delta_A = \overline{A} - A$.
9. Sim simulates the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as follows:
   (a) Sim receives RandCoin from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send γ to them. If Sim receives abort from $\mathcal{A}$, he sends (abort, P_h) to $\mathcal{F}_{\mathsf{patensor}}^k$ and aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.
   (b) Sim follows the protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to compute corrupted parties' shares of $[\sigma]$.
   (c) Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, P_i, \sigma_i, \mathsf{mid}_{\sigma_i})$ from each corrupted party P_i.
   (d) – If MACCheck = 0, Sim computes the P_h's share of $[\sigma]$ based on the corrupted parties' shares (computed by Sim honestly, not the received shares by $\mathcal{F}_{\mathsf{Commit}}$) and $\sum_{i=1}^n \sigma_i = 0$.
   – If MACCheck = 1, Sim sends (abort, P_h) to $\mathcal{F}_{\mathsf{patensor}}^k$ and then samples a random field element as σ. Sim then follows the protocol to compute corrupted parties' shares of $[\sigma]$ and then reconstruct P_h's share.
   (e) For each party corrupted P_i, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties. For the honest party P_h, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_h, \mathsf{mid}_{\sigma_h})$ to all the corrupted parties.
   (f) For each party P_i, Sim receives $(\mathsf{open}, P_i, \mathsf{mid}_{\sigma_i})$ from the corrupted parties and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\sigma_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties.
   (g) Sim follows the protocol to check whether $\sum_{i=1}^n \sigma_i = 0$ holds. If not, Sim aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.

(h) If $\mathsf{MACCheck} = 1$, Sim aborts the simulation.

10. Sim receives $\mathsf{RandCoin}$ from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send r' to them. If Sim receives abort from $\mathcal{A}$, he sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view. Sim then follows the protocol to expand the seed.
11. Sim follows the protocol to compute each corrupted party P_j's shares of $[\![x_\alpha]\!], [\![y_\alpha]\!], [v_\alpha]$, corresponding to each length-k tensor product. Then, Sim samples random elements as P_h's shares of $[x_\alpha + x'_\alpha], [y_\alpha + y'_\alpha]$ and sends them to P_{king} on behalf of P_h. Sim then receives $\overline{x_\alpha + x'_\alpha}, \overline{y_\alpha + y'_\alpha}$ from P_{king}.
12. For each length-k tensor product with index α, Sim locally computes $x_\alpha + x'_\alpha, y_\alpha + y'_\alpha$ with the corrupted parties' shares of $[x_\alpha + x'_\alpha], [y_\alpha + y'_\alpha]$ and P_h's shares of them sampled in the last step. Then, Sim checks whether $\overline{x_\alpha + x'_\alpha} = x_\alpha + x'_\alpha$ and $\overline{y_\alpha + y'_\alpha} = y_\alpha + y'_\alpha$. If not, Sim sets $\mathsf{MACCheck} = 1$.
13. Sim simulates the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as follows:
    (a) Sim receives $\mathsf{RandCoin}$ from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send γ to them. If Sim receives abort from $\mathcal{A}$, he sends (abort, P_h) to $\mathcal{F}^k_{\mathsf{patensor}}$ and aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.
    (b) Sim follows the protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to compute corrupted parties' shares of $[\sigma]$.
    (c) Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, P_i, \sigma_i, \mathsf{mid}_{\sigma_i})$ from each corrupted party P_i.
    (d)
    - If $\mathsf{MACCheck} = 0$, Sim computes the P_h's share of $[\sigma]$ based on the corrupted parties' shares (computed by Sim honestly, not the received shares by $\mathcal{F}_{\mathsf{Commit}}$) and $\sum_{i=1}^n \sigma_i = 0$.
    - If $\mathsf{MACCheck} = 1$, Sim sends (abort, P_h) to $\mathcal{F}^k_{\mathsf{patensor}}$ and samples a random field element as Δ. For each value A_i opened in Π_{Open} protocol:
      * Sim computes $\Delta \cdot A_i$ and computes the honest party's share of $[\Delta \cdot A_i]$ based on the corrupted parties' shares and $\Delta \cdot A_i$. Each A_i is reconstructed based on all the parties' shares, where the corrupted parties' shares are computed by following the protocol.
      * Sim follows the protocol to compute the P_h's share σ_h of $[\sigma]$.
    (e) For each party corrupted P_i, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties. For the honest party P_h, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_h, \mathsf{mid}_{\sigma_h})$ to all the corrupted parties.
    (f) For each party P_i, Sim receives $(\mathsf{open}, P_i, \mathsf{mid}_{\sigma_i})$ from the corrupted parties and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\sigma_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties.
    (g) Sim follows the protocol to check whether $\sum_{i=1}^n \sigma_i = 0$ holds. If not, Sim aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.
    (h) If $\mathsf{MACCheck} = 1$, Sim aborts the simulation.
14. Sim follows the protocol to compute the corrupted parties' shares of $[\tau]$, where the seeds they send to $\mathcal{F}^{\mathsf{prog}}_{\mathsf{tensor}}$ running among two corrupted parties are all set to the seeds Sim sends to them on behalf of $\mathcal{F}_{\mathsf{nVOLE}}$.
15. Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, P_i, \tau_i, \mathsf{mid}_{\tau_i})$ from each corrupted party P_i. Then
    - If $\mathsf{ErrorCheck} = 0$, Sim computes the P_h's share of $[\tau]$ based on the corrupted parties' shares (computed by Sim honestly, not the received shares by $\mathcal{F}_{\mathsf{Commit}}$) and $\sum_{i=1}^n \tau_i = 0$.
    - If $\mathsf{ErrorCheck} = 1$, for each $\alpha = 1, \ldots, m_T$ and $i = 1, \ldots, k$, Sim randomly samples 2 field elements as P_h's shares of $[a_i^{(\alpha)}], [b_i^{(\alpha)}]$. Then, Sim reconstructs $a_i^{(\alpha)}, b_i^{(\alpha)}$ from all parties' shares. Sim then computes $\delta_\alpha = \delta_x^{(\alpha)} \cdot y'_\alpha + \delta_y^{(\alpha)} \cdot x'_\alpha - \sum_{i=1}^k \sum_{j=1}^k r_{a,\alpha}^{(i)'} r_{b,\alpha}^{(j)'} (\delta_{a,i,j} \cdot b_j + \delta_{b,i,j} \cdot a_i)$. Finally, Sim computes the P_h's share of $[\tau]$ based on the corrupted parties' shares (computed by Sim honestly, not the received shares by $\mathcal{F}_{\mathsf{Commit}}$) and the secret $\tau = \sum_{i=1}^n \tau_i = \delta_1 + \cdots + \delta_{m_T}$.
16. For each party corrupted P_i, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_i, \mathsf{mid}_{\tau_i})$ to all the corrupted parties. For the honest party P_h, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_h, \mathsf{mid}_{\tau_h})$ to all the corrupted parties.
17. For each party P_i, Sim receives $(\mathsf{open}, P_i, \mathsf{mid}_{\tau_i})$ from the corrupted parties and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\tau_i, \mathsf{mid}_{\tau_i})$ to all the corrupted parties.
18. Sim follows the protocol to check whether $\sum_{i=1}^n \tau_i = 0$ holds. If not, Sim aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.

19. If $\mathsf{ErrorCheck} = 1$, Sim aborts the simulation. Otherwise, Sim follows the protocol to compute the corrupted parties' outputs of $\mathcal{F}_{\mathsf{patensor}}^{k}$ and sends the outputs to $\mathcal{F}_{\mathsf{patensor}}^{k}$.
20. Sim outputs the adversary's view.

Figure 16: The simulator for the preparation of partially authenticated tensor products.

We construct the following hybrids:

$\mathbf{Hyb}_0$: In this hybrid, Sim runs the protocol honestly on behalf of the honest parties and communicates with $\mathcal{A}$. This corresponds to the real-world scenario.

$\mathbf{Hyb}_1$: In this hybrid, Sim additionally initializes $\mathsf{ErrorCheck} = \mathsf{MACCheck} = 0$ at the beginning. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ have the same output distribution.

$\mathbf{Hyb}_2$: In this hybrid, while emulating $\mathcal{F}_{\mathsf{uatensor}}^{k+1}$, if the received coefficients for linear errors satisfy $\delta_{a,i,j}^{(\alpha)} \neq 0$ or $\delta_{b,i,j}^{(\alpha)} \neq 0$ for any $i, j \in \{1, \ldots, k\}$ and $\alpha \in \{1, \ldots, m_T + \lceil m_T/k \rceil + 1\}$, Sim additionally sets $\mathsf{ErrorCheck} = 1$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

$\mathbf{Hyb}_3$: In this hybrid, for $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$, Sim does not sample the honest party P_h's shares of $[\Delta_a^{(\alpha)}], [\Delta_b^{(\alpha)}]$ and computes P_h's shares of $\{[d_i^{(\alpha)}], [e_i^{(\alpha)}]\}_{i=1}^{k}$ while emulating $\mathcal{F}_{\mathsf{uatensor}}^{k+1}$. Instead, Sim first samples P_h's shares of $[\Delta + \Delta_a^{(\alpha)}], [\Delta + \Delta_b^{(\alpha)}]$ and then computes P_h's shares of $[\Delta_a^{(\alpha)}], [\Delta_b^{(\alpha)}]$ from his share of $[\Delta]$. P_h's shares of $\{[d_i^{(\alpha)}], [e_i^{(\alpha)}]\}_{i=1}^{k}$ are then computed accordingly. Since P_h's shares of $[\Delta_a^{(\alpha)}], [\Delta_b^{(\alpha)}]$ are sampled randomly, his shares of $[\Delta + \Delta_a^{(\alpha)}], [\Delta + \Delta_b^{(\alpha)}]$ are also random. Thus, we only change the order of generating P_h's shares of $[\Delta + \Delta_a^{(\alpha)}], [\Delta + \Delta_b^{(\alpha)}]$ and $[\Delta_a^{(\alpha)}], [\Delta_b^{(\alpha)}]$ without changing their distributions. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same output distribution.

$\mathbf{Hyb}_4$: In this hybrid, for each $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$, after receiving $\overline{\Delta + \Delta_a^{(\alpha)}}, \overline{\Delta + \Delta_b^{(\alpha)}}$ from P_{king}. Sim computes the sum of corrupted parties' shares of $[\Delta]$ from the sum of their shares of $[\Delta_a^{(1)}]$ and $[\Delta + \Delta_a^{(1)}]$, where the sum of their shares of $[\Delta + \Delta_a^{(1)}]$ are computed assuming that P_{king} honestly computes $\Delta + \Delta_a^{(1)} = \overline{\Delta + \Delta_a^{(1)}}$. Sim then reconstructs $\Delta + \Delta_a^{(\alpha)}, \Delta + \Delta_b^{(\alpha)}$ from all parties' shares for each $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ have the same output distribution.

$\mathbf{Hyb}_5$: In this hybrid, Sim additionally computes $\delta_a^{(\alpha)} = \overline{\Delta + \Delta_a^{(\alpha)}} - (\Delta + \Delta_a^{(\alpha)})$ and $\delta_b^{(\alpha)} = \overline{\Delta + \Delta_b^{(\alpha)}} - (\Delta + \Delta_b^{(\alpha)})$. Unless $\nu = \delta_a^{(\alpha)} - \delta_{a,i}^{(\alpha)} = \delta_b^{(\alpha)} - \delta_{b,i}^{(\alpha)}$ and the value are all the same for $\alpha = 1, \ldots, m_T + \lceil m_T/k \rceil + 1$ and $i = 1, \ldots, k$, Sim additionally sets $\mathsf{MACCheck} = 1$. Otherwise, Sim adds ν into the sum of corrupted parties' shares of $[\Delta]$, sets P_{king}'s share to be this sum, and sets all other corrupted parties' shares of $[\Delta]$ to 0. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same output distribution.

$\mathbf{Hyb}_6$: In this hybrid, after emulating $\mathcal{F}_{\mathsf{Coin}}$ to generate r, if for all $i = 1, \ldots, k$, $r_{a,i}^{(m_T + \lceil m_T/k \rceil + 1)} = r_{b,i}^{(m_T + \lceil m_T/k \rceil + 1)} = 0$, Sim aborts the simulation. If each pair of $(r_{a,i}^{(m_T + \lceil m_T/k \rceil + 1)}, r_{b,i}^{(m_T + \lceil m_T/k \rceil + 1)})$ are truly random, then this only happens with probability $|\mathbb{F}|^{-2k}$, which is negligible. Now, we argue that $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ are computationally indistinguishable.

For the sake of contradiction, assume that $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ are computationally distinguishable. By the standard hybrid argument, there exists a PPT algorithm $\mathcal{D}$ that can distinguish the output distributions before and after we change $\mathsf{PRG}(r)$ to a random vector with a non-negligible probability. Now we construct a PPT algorithm $\mathcal{D}'$ that breaks the security of PRG, which is modeled as a PRG. Concretely, $\mathcal{D}'$ receives a string from the challenger, which is either the output of $\mathsf{PRG}(r)$ or a random vector. Then $\mathcal{D}'$ runs Sim with the change that $\mathsf{PRG}(r)$ is replaced by the string received from the challenger. Finally, $\mathcal{D}'$ uses $\mathcal{D}$ to distinguish whether the string received from the challenger is generated from $\mathsf{PRG}(r)$ or sampled randomly. Note that the advantage of $\mathcal{D}'$ is the same as that of $\mathcal{D}$, which breaks the security of PRG. This leads to a contradiction. Thus, the distributions of $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ are computationally indistinguishable.

$\mathbf{Hyb}_7$: In this hybrid, without loss of generality, we assume that $r_{a,k}^{(m_T + \lceil m_T/k \rceil + 1)} \neq 0$, Sim does not sample P_h's share of $[a_k^{(m_T + \lceil m_T/k \rceil + 1)}]$ (and use it to compute P_h's shares of other sharings generated by $\mathcal{F}_{\mathsf{uatensor}}^{k+1}$ related to it) at the beginning. Instead, Sim directly samples P_h's share of $[A]$ and then com-

putes P_h's share of $[a_i^{(m_T+\lceil m_T/k\rceil+1)}]$ from his share of $[A]$ and $\sum_{\alpha=1}^{m_T+\lceil m_T/k\rceil}\sum_{i=1}^{k}(r_{a,i}^{(\alpha)}[a_i^{(\alpha)}]+r_{b,i}^{(\alpha)}[b_i^{(\alpha)}])+\sum_{i=1}^{k-1}(r_{a,i}^{(m_T+\lceil m_T/k\rceil+1)}[a_i^{(m_T+\lceil m_T/k\rceil+1)}]+r_{b,i}^{(m_T+\lceil m_T/k\rceil+1)}[b_i^{(m_T+\lceil m_T/k\rceil+1)}])+r_{b,k}^{(m_T+\lceil m_T/k\rceil+1)}[b_k^{(m_T+\lceil m_T/k\rceil+1)}]$. Since $r_{a,k}^{(m_T+\lceil m_T/k\rceil+1)}\neq 0$ and P_h's share of $[a_k^{(m_T+\lceil m_T/k\rceil+1)}]$ is uniformly random, his share of $[A]$ is also uniformly random. Thus, we only change the order of generating P_h's share of $[a_k^{(m_T+\lceil m_T/k\rceil+1)}]$ and $[A]$ without changing their distributions. Thus, $\mathbf{Hyb}_7$ and $\mathbf{Hyb}_6$ have the same output distribution.

$\mathbf{Hyb}_8$: In this hybrid, while simulating the first execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, if $\mathsf{MACCheck}=0$, Sim does not follow the protocol to compute P_h's share of $[\sigma]$. Instead, Sim computes the P_h's share of $[\sigma]$ based on the corrupted parties' shares and $\sum_{i=1}^{n}\sigma_i=0$. Note that in this case, we can regard that the currently computed corrupted parties' shares of $[\Delta]$ are what they sampled, and they all follow the protocol with each $\delta_{a,i}^{(\alpha)}=\delta_{b,i}^{(\alpha)}=0$, so it must hold that $[\sigma]=[0]$. Thus, we only change the way of generating P_h's share of $[\sigma]$ without changing its distribution. Thus, $\mathbf{Hyb}_8$ and $\mathbf{Hyb}_7$ have the same output distribution.

$\mathbf{Hyb}_9$: In this hybrid, while simulating the first execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, if $\mathsf{MACCheck}=1$, Sim doesn't follow the protocol to compute the honest party P_h's share of $[\sigma]$. Instead, Sim samples it directly. Then, Sim samples a random field element as P_h's share of Δ. Moreover, Sim does not follow the protocol to sample P_h's shares of $[a_i^{(\alpha)}],[b_i^{(\alpha)}]$ for $i=1,\ldots,k,\alpha=1,\ldots,m_T+\lceil m_T/k\rceil$. Note that the error on $[\sigma]=A\cdot[\Delta]-[A\cdot\Delta]$ is

$$\delta_A\cdot\Delta-\sum_{\alpha=1}^{m_T+\lceil m_T/k\rceil+1}\sum_{i=1}^{k}(r_{a,i}^{(\alpha)}(\delta_b^{(\alpha)}-\delta_{b,i}^{(\alpha)})\cdot a_i^{(\alpha)}+r_{b,i}^{(\alpha)}(\delta_a^{(\alpha)}-\delta_{a,i}^{(\alpha)})\cdot b_i^{(\alpha)}).$$

Sim samples these shares after $\sigma,\delta_A,\{\delta_a^{(\alpha)},\delta_b^{(\alpha)},\{\delta_{a,i}^{(\alpha)},\delta_{b,i}^{(\alpha)}\}_{i=1}^{k}\}_{\alpha=1}^{m_T+\lceil m_T/k\rceil+1}$ are determined based on the above equation. Since $\mathsf{MACCheck}=1$ only happens when $\delta_a^{(\alpha)}-\delta_{a,i}^{(\alpha)},\delta_b^{(\alpha)}-\delta_{b,i}^{(\alpha)}$ are not all the same, when each $a_i^{(\alpha)}$ or $b_i^{(\alpha)}$ for $i=1,\ldots,k,\alpha=1,\ldots,m_T+\lceil m_T/k\rceil$ is randomly sampled, the error is also random given A and all the coefficients on them. Note that all of these $a_i^{(\alpha)},b_i^{(\alpha)}$ are used for computing any message sent to $\mathcal{A}$ before the error coefficients $\delta_a^{(\alpha)}-\delta_{a,i}^{(\alpha)},\delta_b^{(\alpha)}-\delta_{b,i}^{(\alpha)}$ are determined, they can be sampled after the coefficients are received. Then, the error on $[\sigma]$ is random, so sampling σ randomly and then sampling P_h's shares of $[a_i^{(\alpha)}],[b_i^{(\alpha)}]$ for $i=1,\ldots,k,\alpha=1,\ldots,m_T+\lceil m_T/k\rceil$ based on σ does not change the output distribution. Thus, $\mathbf{Hyb}_9$ and $\mathbf{Hyb}_8$ have the same output distribution.

$\mathbf{Hyb}_{10}$: In this hybrid, while simulating the first execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, after following the protocol to check whether $\sum_{i=1}^{n}\sigma_i=0$, Sim aborts the simulation if $\mathsf{MACCheck}=1$. The only difference between $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ is that the corrupted parties may guess σ correctly and attach additive errors on their committed shares of $[\sigma]$ to make the opened secret 0. This may lead the check to pass even when $\mathsf{MACCheck}=1$. However, by the randomness of σ, this only happens at most with probability $1/|\mathbb{F}|$, which is negligible. Thus, the distributions of $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ are statistically close.

$\mathbf{Hyb}_{11}$: In this hybrid, for each length-k tensor product (with index α), Sim does not generates P_h's shares of $[\![x'_\alpha]\!],[\![y'_\alpha]\!]$ and then follows the protocol to compute $[\![x_\alpha+x'_\alpha]\!],[\![y_\alpha+y'_\alpha]\!]$. Instead, Sim samples random elements as P_h's shares of $[x_\alpha+x'_\alpha],[y_\alpha+y'_\alpha]$ randomly and then use them to compute his shares of $\boldsymbol{a}_\alpha^{(h)},\boldsymbol{b}_\alpha^{(h)}$ and samples P_i's shares of $[x'_\alpha],[y'_\alpha]$. P_h's shares of $[\Delta\cdot x'_\alpha],[\Delta\cdot y'_\alpha]$ are then computed accordingly from $\Delta,x'_\alpha,y'_\alpha$. Since P_h's shares of $[x'_\alpha],[y'_\alpha]$ are sampled randomly and are added into P_h's shares of $[x_\alpha+x'_\alpha],[y_\alpha+y'_\alpha]$, so P_h's shares of $[x_\alpha+x'_\alpha],[y_\alpha+y'_\alpha]$ is also random. Therefore, we only change the order of generating P_h's shares of $[x'_\alpha],[y'_\alpha]$ and $[x_\alpha+x'_\alpha],[y_\alpha+y'_\alpha]$ without changing their distributions. Thus, $\mathbf{Hyb}_{11}$ and $\mathbf{Hyb}_{10}$ have the same output distribution.

$\mathbf{Hyb}_{12}$: In this hybrid, for each length-k tensor product (with index α), Sim locally computes $x_\alpha+x'_\alpha,y_\alpha+y'_\alpha$ with the corrupted parties' shares of $[x_\alpha+x'_\alpha],[y_\alpha+y'_\alpha]$ and P_h's shares of them sampled in the last hybrid. Then, Sim additionally checks whether $\overline{x_\alpha+x'_\alpha}=x_\alpha+x'_\alpha$ and $\overline{y_\alpha+y'_\alpha}=y_\alpha+y'_\alpha$. If not, Sim sets $\mathsf{MACCheck}=1$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{12}$ and $\mathbf{Hyb}_{11}$ have the same output distribution.

$\mathbf{Hyb}_{13}$: In this hybrid, while simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, if $\mathsf{MACCheck}=0$, Sim does not follow the protocol to compute P_h's share of $[\sigma]$. Instead, Sim computes the P_h's share of $[\sigma]$ based on the corrupted parties' shares and $\sum_{i=1}^{n}\sigma_i=0$. Note that when $\mathsf{MACCheck}=0$, all the opened values are opened

correctly via Π_{Open}. In this case, it must hold that $[\sigma] = [0]$, so we only change the way of generating P_h's share of $[\sigma]$ without changing its distribution. Thus, $\mathbf{Hyb}_{13}$ and $\mathbf{Hyb}_{12}$ have the same output distribution.

$\mathbf{Hyb}_{14}$: In this hybrid, while simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, if $\mathsf{MACCheck} = 1$, Sim doesn't directly compute the honest party P_h's share of $[\Delta \cdot A_i]$ for each opened value A_i. Instead, Sim first computes each $\Delta \cdot A_i$, and then Sim computes the honest party's share of $[\Delta \cdot A_i]$ based on the corrupted parties' shares and $\Delta \cdot A_i$. We just change the way the honest party's share of each $[\Delta \cdot A_i]$ is generated without changing its distribution. Thus, $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{13}$ have the same output distribution.

$\mathbf{Hyb}_{15}$: In this hybrid, while simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, after following the protocol to check whether $\sum_{i=1}^{n} \sigma_i = 0$, Sim aborts the simulation if $\mathsf{MACCheck} = 1$. The only difference between $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$ is that the corrupted parties may not open all the values of SPDZ sharing checked in this execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ correctly, but the verification of MACs may pass.

We suppose that $A_1, \ldots, A_m$ are opened with additive errors $\delta^{(1)}, \ldots, \delta^{(m)}$, and the sum of all the committed shares $[\sigma]$ of corrupted parties is with an additive error δ_σ. Then

$$\begin{aligned}\sigma =& \sum_{i\in[1,n], i\neq h} \left(\sum_{\alpha=1}^{m} \chi^\alpha \cdot (\Delta \cdot A_\alpha)^{(i)} - \Delta_i \cdot \sum_{\alpha=1}^{m} \chi^\alpha \cdot A_\alpha \right) + \delta_\sigma \\ &+ \left(\sum_{\alpha=1}^{m} \chi^\alpha \cdot (\Delta \cdot A_\alpha)^{(h)} - \Delta_h \cdot \sum_{\alpha=1}^{m} \chi^\alpha \cdot (A_\alpha + \delta^{(\alpha)}) \right) \\ =& \delta_\sigma + \Delta_h \cdot \sum_{\alpha=1}^{m} \chi^\alpha \cdot \delta^{(\alpha)}.\end{aligned}$$

Here $(\Delta \cdot A_\alpha)^{(i)}$ is used to denote P_i's share of $[\Delta \cdot A_\alpha]$. Since Δ_h is not used in computing any message sent to the corrupted parties at this moment, $\Delta_h \cdot \sum_{\alpha=1}^{m} \chi^\alpha \cdot \delta^{(\alpha)}$ is a random element in $\mathbb{F}$ as long as $\sum_{\alpha=1}^{m} \chi^\alpha \cdot \delta^{(\alpha)}$ is non-zero. When $(\delta^{(1)}, \ldots, \delta^{(m)})$ is a nonzero vector, since a non-zero degree-m polynomial only has at most m zeros, so $\sum_{\alpha=1}^{m} \chi^\alpha \cdot \delta^{(\alpha)} = 0$ only happens with at most $m/|\mathbb{F}|$ probability. Since $m = \mathsf{poly}(n, \kappa)$, the probability is negligible. On the other hand, in the case where $\sum_{\alpha=1}^{m} \chi^\alpha \cdot \delta^{(\alpha)} = 0$, σ is opened to a random element in $\mathbb{F}$, which equals 0 with $1/|\mathbb{F}|$ probability, which is also negligible.

Thus, the distributions of $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$ are statistically close.

$\mathbf{Hyb}_{16}$: In this hybrid, after simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ (so $\mathsf{MACCheck} = 0$ must hold), if $\mathsf{ErrorCheck} = 0$, Sim does not follow the protocol to compute P_h's share of $[\tau]$. Instead, Sim computes the P_h's share of $[\tau]$ based on the corrupted parties' shares and $\sum_{i=1}^{n} \tau_i = 0$. Note that when $\mathsf{ErrorCheck} = 0$ and $\mathsf{MACCheck} = 0$, the corrupted parties open all the values honestly and interact honestly with $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}$ with P_h. Therefore, the length-k tensor products are correctly computed. In this case, it must hold that $[\tau] = [0]$, so we only change the way of generating P_h's share of $[\tau]$ without changing its distribution. Thus, $\mathbf{Hyb}_{16}$ and $\mathbf{Hyb}_{15}$ have the same output distribution.

$\mathbf{Hyb}_{17}$: In this hybrid, after simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ (so $\mathsf{MACCheck} = 0$ must hold), if $\mathsf{ErrorCheck} = 1$, Sim does not follow the protocol to compute P_h's share of $[\tau]$. Instead, Sim computes $\delta_\alpha = \delta_x^{(\alpha)} \cdot y'_\alpha + \delta_y^{(\alpha)} \cdot x'_\alpha - \sum_{i=1}^{k} \sum_{j=1}^{k} r_{a,\alpha}^{(i)'} r_{b,\alpha}^{(j)'} (\delta_{a,i,j} \cdot b_j + \delta_{b,i,j} \cdot a_i)$. Finally, Sim computes the P_h's share of $[\tau]$ based on the corrupted parties' shares (computed by Sim honestly, not the received shares by $\mathcal{F}_{\mathsf{Commit}}$) and the secret $\tau = \sum_{i=1}^{n} \tau_i = \delta_1 + \cdots + \delta_{m_T}$.

Note that, compared to the case of $\mathsf{ErrorCheck} = 0$ where corrupted parties honestly send their seeds to $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}$, what Sim computes is just the attached error on τ. Therefore, we only change the way of computing τ without changing its distribution.

Thus, $\mathbf{Hyb}_{17}$ and $\mathbf{Hyb}_{16}$ have the same output distribution.

$\mathbf{Hyb}_{18}$: In this hybrid, after following the protocol to check whether $\sum_{i=1}^{n} \tau_i = 0$, Sim aborts the simulation if $\mathsf{ErrorCheck} = 1$. The only difference between $\mathbf{Hyb}_{18}$ and $\mathbf{Hyb}_{17}$ is that corrupted parties may attach non-zero linear errors on a $[c_{i,j}^{(\alpha)}]$, but the verification of linear errors may pass, i.e., the additional error δ attached to τ when corrupted parties commit to their shares of $[\tau]$ matches the error $\delta_1 + \cdots + \delta_{m_T}$ attached to P_h's share of $[\tau]$.

Without loss of generality, we assume that the $\delta_{a,i,j}^{(\alpha)} \neq 0$. Then, an error of $r_{a,\alpha}^{(i)'} r_{b,\alpha}^{(j)'} \delta_{a,i,j}^{(\alpha)} \cdot b_j^{(\alpha)}$ is attached to $\delta_1 + \cdots + \delta_{m_T}$. As long as $r_{a,\alpha}^{(i)'}$ is uniformly random in $\mathbb{F}$ and $r_{b,\alpha}^{(j)'}$ is non-zero, the total coefficient on $b_j^{(\alpha)}$ attached to the error is $\sum_{i=1}^{k} r_{a,\alpha}^{(i)'} r_{b,\alpha}^{(j)'} \delta_{a,i,j}^{(\alpha)}$, which is also uniformly random (as $r_a^{(i)} \delta_{a,i,j}^{(\alpha)}$ is random). Therefore, the total error term is uniformly random in $\mathbb{F}$ based on the randomness of $b_j^{(\alpha)}$. In this case, the total error $\delta_1 + \cdots + \delta_{m_T}$ is also uniformly random in $\mathbb{F}$, so the probability that $\delta_1 + \cdots + \delta_{m_T} = \delta$ is no more than $1/|\mathbb{F}|$, which is negligible.

Note that if $r_{a,\alpha}^{(i)'}, r_{b,\alpha}^{(j)'}$ are truly random, then $r_{b,\alpha}^{(j)'}$ is non-zero with an overwhelming probability. Since $r_{a,\alpha}^{(i)'}, r_{b,\alpha}^{(j)'}$ are generated from a PRG with a truly random seed, if the two hybrids are computationally distinguishable, then the PRG can be broken due to the same reason as in $\mathbf{Hyb}_6$. Therefore, the distribution only changes with a negligible probability.

Thus, the distributions of $\mathbf{Hyb}_{18}$ and $\mathbf{Hyb}_{17}$ are computationally indistinguishable.

$\mathbf{Hyb}_{19}$: In this hybrid, when emulating $\mathcal{F}_{\mathsf{uatensor}}^{k+1}$, Sim delays the generation of the P_h's shares when they are needed. This does not change the output distribution. Thus, $\mathbf{Hyb}_{19}$ and $\mathbf{Hyb}_{18}$ have the same output distribution.

Note that if the protocol does not abort at the end of the execution and Sim does not abort the simulation, the honest party's shares are not used during the interaction with corrupted parties. In this case, by construction, corrupted parties honestly follow the protocol and Sim only need the honest party's shares from $\mathcal{F}_{\mathsf{uatensor}}^{k+1}$ to prepare the honest party's output of $\mathcal{F}_{\mathsf{patensor}}^{k}$.

$\mathbf{Hyb}_{20}$: In this hybrid, we change the preparation of the honest party's output when the protocol does not abort at the end of the execution and Sim does not abort the simulation. Sim no longer generates the P_h's output shares when emulating $\mathcal{F}_{\mathsf{uatensor}}^{k+1}$. Instead, for each $[\![r]\!]$ (including each $[\![a_i]\!], [\![b_i]\!]$ in the length-k tensor products), Sim samples a random value as r and then computes the shares of honest parties using the secret r, the MAC key Δ, and the shares of corrupted parties. By the correctness of the construction, the distributions of $\mathbf{Hyb}_{20}$ and $\mathbf{Hyb}_{19}$ are identical.

$\mathbf{Hyb}_{21}$: In this hybrid, at the end of the simulation, Sim sends the corrupted parties' outputs of $\mathcal{F}_{\mathsf{prep}}$ and lets $\mathcal{F}_{\mathsf{patensor}}^{k}$ compute the honest party's output. Since the process of generating Δ by Sim himself and by $\mathcal{F}_{\mathsf{patensor}}^{k}$ is the same, this doesn't change the output distribution. Thus, $\mathbf{Hyb}_{21}$ and $\mathbf{Hyb}_{20}$ have the same output distribution.

Note that $\mathbf{Hyb}_{21}$ is the ideal-world scenario. Thus, $\Pi_{\mathsf{patensor}}^{k}$ computes $\mathcal{F}_{\mathsf{patensor}}^{k}$ with computational security.

D Security Proof for the Preparation of Unauthenticated Tensor Products

We provide the security proof of protocol $\Pi_{\mathsf{uatensor}}^{k}$ instantiated from 2-party tensor product correlations and n-party unauthenticated triples, respectively.

D.1 Instantiation from 2-party tensor product correlations

We prove that the protocol $\Pi_{\mathsf{uatensor}}^{k}$ (Figure 5) securely computes $\mathcal{F}_{\mathsf{uatensor}}^{k}$ in the $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$-hybrid model here. We prove the security by constructing an ideal adversary Sim. Then we will show that the output in the ideal world is indistinguishable from that in the real world against the adversary $\mathcal{A}$ using hybrid arguments. Since the parties have no inputs to $\mathcal{F}_{\mathsf{uatensor}}^{k}$, and the outputs of corrupted parties are fully determined by Sim in the ideal world (so there are no restrictions on honest parties' outputs other than one honest party's), we can simply consider the maximum corruption case, where there is only one honest party P_h. If another party is honest, Sim just needs to honestly emulate him to run the protocol.

Simulator Sim

1. For the invocations of $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$ between P_h and a corrupted party P_i, Sim emulates $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$ to receive $s_a^{(i)}, s_b^{(i)}$ and P_i's output of $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$ from Sim. If abort is received, Sim sends (abort, P_h) to $\mathcal{F}_{\mathsf{uatensor}}^{k}$. After completing the simulation, Sim outputs the adversary's view. Then, Sim emulates $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$ to send P_i's output to P_i. For other invocations of $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$, Sim faithfully emulates $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$.
2. If the protocol is not aborted by P_h, Sim assumes that each corrupted party P_i uses $s_a^{(i)}, s_b^{(i)}$ honestly during all the invocations of $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$. Then, Sim follows the protocol to compute corrupted parties' outputs of $\mathcal{F}_{\mathsf{uatensor}}^{k}$. Sim sends them to $\mathcal{F}_{\mathsf{uatensor}}^{k}$.
3. Sim sets $\delta_{a,i,j} = \delta_{b,i,j} = 0$ for all the unauthenticated tensor products and sends them to $\mathcal{F}_{\mathsf{uatensor}}$.
4. Sim outputs the adversary's view

Figure 17: The simulator for the preparation of unauthenticated tensor products from 2-party tensor product correlations.

We construct the following hybrids:

$\mathbf{Hyb}_0$: In this hybrid, Sim runs the protocol honestly on behalf of the honest parties and communicates with $\mathcal{A}$. This corresponds to the real-world scenario.

$\mathbf{Hyb}_1$: In this hybrid, Sim extracts each corrupted party P_i's input seeds $s_a^{(i)}, s_b^{(i)}$ during the invocation of $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$ between P_h and P_i, Sim emulates $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$ to receive $s_a^{(i)}, s_b^{(i)}$ and P_i's output of $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$ from Sim. Then, Sim still emulates $\mathcal{F}_{\mathsf{tensor}}^{\mathsf{prog}}(m_U \cdot k, k)$ faithfully for the invocations between two corrupted parties, but Sim regards that P_i always uses $s_a^{(i)}, s_b^{(i)}$ as his input seeds to compute corrupted parties' outputs. This does not affect the output distribution. Thus, $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ have the same distribution.

$\mathbf{Hyb}_2$: In this hybrid, P_h does not follow the protocol to compute his output. Instead, Sim sends corrupted parties' outputs to $\mathcal{F}_{\mathsf{uatensor}}^{k}$, sets $\delta_{a,i,j} = \delta_{b,i,j} = 0$ for all the unauthenticated tensor products, and sends them to $\mathcal{F}_{\mathsf{uatensor}}^{k}$. Honest parties obtain their outputs from $\mathcal{F}_{\mathsf{uatensor}}^{k}$. We only change how honest parties' outputs are generated, where the only difference from $\mathbf{Hyb}_1$ is that the honest parties' shares of $[\boldsymbol{a}], [\boldsymbol{b}]$ become truly random rather than expanded from a PRG. Now, we argue that $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ are computationally indistinguishable.

For the sake of contradiction, assume that $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ are computationally distinguishable. By the standard hybrid argument, there exists a PPT algorithm $\mathcal{D}$ that can distinguish the output distributions before and after we change $\mathsf{Expand}_{m_U \cdot k,k}(s_a^{(h)}), \mathsf{Expand}_{m_U \cdot k,k}(s_b^{(h)})$ to random vectors with a non-negligible probability. Now we construct a PPT algorithm $\mathcal{D}'$ that breaks the security of $\mathsf{Expand}_{m_U \cdot k,k}$, which is modeled as a PRG. Concretely, $\mathcal{D}'$ receives a string from the challenger, which is either the output of $\mathsf{Expand}_{m_U \cdot k,k}(s_a^{(h)}), \mathsf{Expand}_{m_U \cdot k,k}(s_b^{(h)})$ or random vectors. Then $\mathcal{D}'$ runs Sim with the change that $\mathsf{Expand}_{m_U \cdot k,k}(s_a^{(h)})$ and $\mathsf{Expand}_{m_U \cdot k,k}(s_b^{(h)})$ are replaced by the string received from the challenger. Finally, $\mathcal{D}'$ uses $\mathcal{D}$ to distinguish whether the string received from the challenger is generated from $\mathsf{Expand}_{m_U \cdot k,k}(s_a^{(h)})$ and $\mathsf{Expand}_{m_U \cdot k,k}(s_b^{(h)})$ or sampled randomly. Note that the advantage of $\mathcal{D}'$ is the same as that of $\mathcal{D}$, which breaks the security of $\mathsf{Expand}_{m_U \cdot k,k}$. This leads to a contradiction. Thus, the distributions of $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ are computationally indistinguishable.

Note that $\mathbf{Hyb}_2$ is the ideal-world scenario. Thus, $\Pi_{\mathsf{uatensor}}^{k}$ computes $\mathcal{F}_{\mathsf{uatensor}}^{k}$ with computational security.

D.2 Instantiation from n-party unauthenticated triples

We provide the protocol $\Pi_{\mathsf{uatensor}}^{k}$ for instantiating $\mathcal{F}_{\mathsf{uatensor}}^{k}$ [BCG+20] below.

Protocol $\Pi^k_{\mathsf{uatensor}}$

Public Parameter: The number m_U of unauthenticated tensor products we need to generate.

1. Each party P_i samples $2k$ random seeds $s^{(i)}_{a,1}, \ldots, s^{(i)}_{a,k}, s^{(i)}_{b,1}, \ldots, s^{(i)}_{b,k}$ in S, where each $\mathsf{Expand}^{(m_U)}(s^{(i)}_{a,\alpha})$ (resp. $\mathsf{Expand}^{(m_U)}(s^{(i)}_{b,\alpha})$) serves as the vector of P_i's shares of $[a_\alpha]$ (resp. $[b_\alpha]$) in all the unauthenticated tensor products. We denote the sharing of $[a_\alpha]$ (resp. $[b_\alpha]$) in all the unauthenticated tensor products by $[\boldsymbol{a}_\alpha]$ (resp. $[\boldsymbol{b}_\alpha]$). If abort is received, the party aborts the protocol.
2. For each pair of $(\alpha, \beta) \in \{1, \ldots, k\}^2$, each party P_i sends $(\mathsf{UAtrip}, s^{(i)}_{a,\alpha}, s^{(i)}_{b,\beta})$ to $\mathcal{F}^{\mathsf{prog}}_{\mathsf{uatrip}}(m_U)$. Then, the parties receive their shares of $[\boldsymbol{a}_\alpha * \boldsymbol{b}_\beta]$, which contains $[c_{\alpha,\beta}]$ in each unauthenticated tensor product. Finally, the parties obtain all the m_U tensor products in the form of $\{([a_i], [b_i])\}^k_{i=1}, \{[c_{i,j}]\}^k_{i,j=1}$ as their outputs.

Figure 18: Generating unauthenticated tensor products from unauthenticated Beaver triples.

We prove that the protocol $\Pi^k_{\mathsf{uatensor}}$ (Figure 18) securely computes $\mathcal{F}^k_{\mathsf{uatensor}}$ in the $\mathcal{F}^{\mathsf{prog}}_{\mathsf{uatrip}}(m_U)$-hybrid model here. We prove the security by constructing an ideal adversary Sim. Then we will show that the output in the ideal world is indistinguishable from that in the real world against the adversary $\mathcal{A}$ using hybrid arguments. Since the parties have no inputs to $\mathcal{F}^k_{\mathsf{uatensor}}$, and the outputs of corrupted parties are fully determined by Sim in the ideal world (so there are no restrictions on honest parties' outputs other than one honest party's), we can simply consider the maximum corruption case, where there is only one honest party P_h. If another party is honest, Sim just needs to honestly emulate him to run the protocol.

Simulator Sim

1. For the invocations of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{uatrip}}(m_U)$ corresponding to $\alpha = 1$, Sim emulates $\mathcal{F}^{\mathsf{prog}}_{\mathsf{uatrip}}(m_U)$ to receive $s^{(i)}_{b,\beta}$ for each corrupted party P_i and each $\beta = 1, \ldots, k$. Similarly, for the invocations of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{uatrip}}(m_U)$ corresponding to $\beta = 1$, Sim emulates $\mathcal{F}^{\mathsf{prog}}_{\mathsf{uatrip}}(m_U)$ to receive $s^{(i)}_{a,\alpha}$ for each corrupted party P_i and each $\alpha = 1, \ldots, k$.
2. For each pair of $(\alpha, \beta) \in \{1, \ldots, k\}^2$, Sim emulates $\mathcal{F}^{\mathsf{prog}}_{\mathsf{uatrip}}(m_U)$ to receive $\overline{s^{(i)}_{a,\alpha}}, \overline{s^{(i)}_{b,\beta}}$ from each corrupted party P_i. If (abort, P_h) is received, Sim sends (abort, P_h) to $\mathcal{F}^k_{\mathsf{uatensor}}$ and aborts the protocol on behalf of P_h. After completing the simulation, Sim outputs the adversary's view. Otherwise, Sim computes $\boldsymbol{\delta}_{a,\alpha,\beta} = \sum_{P_i \in \mathcal{C}} (\mathsf{Expand}^{(m_U)}(\overline{s^{(i)}_{a,\alpha}}) - \mathsf{Expand}^{(m_U)}(s^{(i)}_{a,\alpha}))$ (resp. $\boldsymbol{\delta}_{b,\alpha,\beta} = \sum_{P_i \in \mathcal{C}} (\mathsf{Expand}^{(m_U)}(\overline{s^{(i)}_{b,\beta}}) - \mathsf{Expand}^{(m_U)}(s^{(i)}_{b,\beta}))$) as the vector of $\delta_{a,\alpha,\beta}$ (resp. $\delta_{b,\alpha,\beta}$) for all m_U unauthenticated tensor products. Sim then sends $\{\delta_{a,\alpha,\beta}, \delta_{b,\alpha,\beta}\}_{i,j \in \{1,\ldots,k\}}$ for each unauthenticated tensor product to $\mathcal{F}^k_{\mathsf{uatensor}}$.
3. Sim regards that each corrupted party P_i uses $s^{(i)}_{a,1}, \ldots, s^{(i)}_{a,k}, s^{(i)}_{b,1}, \ldots, s^{(i)}_{b,k}$ to compute his shares of $\{[a_i], [b_i]\}^k_{i=1}$ in each unauthenticated tensor product. Then, Sim sends the corrupted parties' shares of them to $\mathcal{F}^k_{\mathsf{uatensor}}$.
4. For each unauthenticated tensor product and each pair of $(\alpha, \beta) \in \{1, \ldots, k\}^2$, Sim follows the protocol to compute corrupted parties' shares of $[c_{\alpha,\beta}]$. Then, Sim adds $\delta_{a,\alpha,\beta} \cdot \delta_{b,\alpha,\beta}$ on a corrupted party's share of $[c_{\alpha,\beta}]$. Finally, Sim sends the corrupted parties' shares of $[c_{\alpha,\beta}]$ to $\mathcal{F}^k_{\mathsf{uatensor}}$.
5. Sim outputs the adversary's view.

Figure 19: The simulator for the preparation of unauthenticated tensor products from n-party unauthenticated triples.

We construct the following hybrids:

$\mathbf{Hyb}_0$: In this hybrid, Sim runs the protocol honestly on behalf of the honest parties and communicates with $\mathcal{A}$. This corresponds to the real-world scenario.

$\mathbf{Hyb}_1$: In this hybrid, Sim extracts each corrupted party P_i's input seeds $\{s^{(i)}_{b,\beta}\}^k_{\beta=1}$ during the invocations of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{uatrip}}(m_U)$ corresponding to $\alpha = 1$. Similarly, Sim extracts each corrupted party P_i's input seeds $\{s^{(i)}_{a,\alpha}\}^k_{\alpha=1}$ during the invocations of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{uatrip}}(m_U)$ corresponding to $\beta = 1$. Sim uses these seeds to compute corrupted parties' shares of $\{[a_i], [b_i]\}^k_{i=1}$ in each unauthenticated tensor product. This does not affect the output

distribution. Thus, $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ have the same distribution.

$\mathbf{Hyb}_2$: In this hybrid, for each pair of $(\alpha, \beta) \in \{1, \ldots, k\}^2$, after emulating $\mathcal{F}_{\mathsf{uatrip}}^{\mathsf{prog}}(m_U)$ to receive $\overline{s_{a,\alpha}^{(i)}}, \overline{s_{b,\beta}^{(i)}}$ from each corrupted party P_i. If (abort, P_h) is not received, Sim additionally computes $\boldsymbol{\delta}_{a,\alpha,\beta} = \sum_{P_i \in \mathcal{C}} (\mathsf{Expand}^{(m_U)}(\overline{s_{a,\alpha}^{(i)}}) - \mathsf{Expand}^{(m_U)}(s_{a,\alpha}^{(i)}))$ (resp. $\boldsymbol{\delta}_{b,\alpha,\beta} = \sum_{P_i \in \mathcal{C}} (\mathsf{Expand}^{(m_U)}(\overline{s_{b,\beta}^{(i)}}) - \mathsf{Expand}^{(m_U)}(s_{b,\beta}^{(i)}))$) as the vector of $\delta_{a,\alpha,\beta}$ (resp. $\delta_{b,\alpha,\beta}$) for all m_U unauthenticated tensor products. This does not affect the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same distribution.

$\mathbf{Hyb}_3$: In this hybrid, for each unauthenticated tensor product and each pair of $(\alpha, \beta) \in \{1, \ldots, k\}^2$, Sim follows the protocol to compute corrupted parties' shares of $[c_{\alpha,\beta}]$. Then, Sim adds $\delta_{a,\alpha,\beta} \cdot \delta_{b,\alpha,\beta}$ on a corrupted party's share of $[c_{\alpha,\beta}]$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same distribution.

$\mathbf{Hyb}_4$: In this hybrid, Sim does not generate P_h's shares of $\{[a_i], [b_i]\}_{i=1}^k$ in each unauthenticated tensor product via the expansion algorithm. Instead, Sim samples random field elements as them. Similarly, while emulating $\mathcal{F}_{\mathsf{uatrip}}^{\mathsf{prog}}(m_U)$, Sim uses the sampled shares to compute P_h's output. Now, we argue that $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ are computationally indistinguishable.

For the sake of contradiction, assume that $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ are computationally distinguishable. By the standard hybrid argument, there exists a PPT algorithm $\mathcal{D}$ that can distinguish the output distributions before and after we change P_h's expanded shares to random vectors with a non-negligible probability. Now we construct a PPT algorithm $\mathcal{D}'$ that breaks the security of $\mathsf{Expand}^{(m_U)}$, which is modeled as a PRG. Concretely, $\mathcal{D}'$ receives a string from the challenger, which is either the outputs of $\mathsf{Expand}^{(m_U)}$ or random vectors. Then $\mathcal{D}'$ runs Sim with the change that the expanded shares are replaced by the string received from the challenger. Finally, $\mathcal{D}'$ uses $\mathcal{D}$ to distinguish whether the string received from the challenger is generated from $\mathsf{Expand}^{(m_U)}$ or sampled randomly. Note that the advantage of $\mathcal{D}'$ is the same as that of $\mathcal{D}$, which breaks the security of $\mathsf{Expand}^{(m_U)}$. This leads to a contradiction. Thus, the distributions of $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ are computationally indistinguishable.

$\mathbf{Hyb}_5$: In this hybrid, P_h does not follow the protocol to compute his output. Instead, Sim sends corrupted parties' outputs to $\mathcal{F}_{\mathsf{uatensor}}^k$, sends $\{\delta_{a,\alpha,\beta}, \delta_{b,\alpha,\beta}\}_{i,j=1}^k$ for each unauthenticated tensor product to $\mathcal{F}_{\mathsf{uatensor}}^k$. Honest parties obtain their outputs from $\mathcal{F}_{\mathsf{uatensor}}^k$. We only change how honest parties' outputs are generated. For each pair of (α, β), compared to the honest execution where each corrupted party P_i uses seeds $s_{a,\alpha}^{(j)}, s_{b,\beta}^{(i)}$ in the invocation of $\mathcal{F}_{\mathsf{uatrip}}^{\mathsf{prog}}(m_U)$ where the parties obtain $[a_\alpha \cdot b_\beta]$, the parties actually obtain $[(a_\alpha + \delta_{a,\alpha,\beta}) \cdot (b_\beta + \delta_{b,\alpha,\beta})]$. Note that we add $\delta_{a,\alpha,\beta} \cdot \delta_{b,\alpha,\beta}$ on a corrupted party's share and now P_h's share is computed to make the sharing $[a_\alpha \cdot b_\beta + \delta_{b,\alpha,\beta} \cdot a_\alpha + \delta_{a,\alpha,\beta} \cdot b_\alpha]$, so P_h's share remain unchanged. Therefore, we only change the way of computing P_h's output without changing its distribution. Thus, $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same distribution.

Note that $\mathbf{Hyb}_5$ is the ideal-world scenario. Thus, $\Pi_{\mathsf{uatensor}}^k$ computes $\mathcal{F}_{\mathsf{uatensor}}^k$ with computational security.

E Regular Dual-LPN Assumption

We provide the formal definition of the regular dual-LPN assumption here.

Definition 1 (d-Regular Vector). *A d-regular vector $\boldsymbol{e}$ is defined as a set of d vectors $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_d$ concatenated, wherein each $\boldsymbol{e}_i$ is a sparse vector with Hamming weight one.*

Definition 2 (Regular Dual-LPN Assumption). *Let $H \in \mathbb{F}^{\gamma \times \gamma'}$, and consider the following game $G_b(\kappa)$ with a PPT adversary $\mathcal{A}$, parameterised by a bit b and the security parameter κ:*

1. *Sample a random, d-regular vector $\boldsymbol{e} \in \mathbb{F}^{\gamma'}$.*
2. *If $b = 1$, let $\boldsymbol{y} = H \cdot \boldsymbol{e}$, otherwise sample $\boldsymbol{y} \leftarrow \mathbb{F}^{\gamma}$.*
3. *Send $\boldsymbol{y}$ to $\mathcal{A}$, which then outputs a bit b' (in case of abort, define the output of $\mathcal{A}$ to be $\perp$).*

The assumption states that $\mathcal{A}$ has a negligible advantage in distinguishing $G_0(\kappa)$ and $G_1(\kappa)$.

F Preprocessing Functionality

The functionality $\mathcal{F}_{\mathsf{prep}}$ for the preprocessing phase of our main protocol is presented as follows in Figure 20.

Functionality $\mathcal{F}_{\mathsf{prep}}$

Init: On receiving $(\mathsf{Prep}, m_R, m_P, m_A, m_T)$ from all the honest parties:

1. The trusted party samples a random element in $\mathbb{F}$ as Δ.
2. The trusted party receives corrupted parties' shares of $[\Delta]$ from Sim and samples the honest parties' shares of $[\Delta]$ based on Δ and the corrupted parties' shares of $[\Delta]$.
3. The trusted party distributes $[\Delta]$ to the parties.

Random Values: Repeat the following m_R times:

1. The trusted party randomly samples $r \in \mathbb{F}$ and computes $\Delta \cdot r$.
2. The trusted party receives corrupted parties' shares of $[\![r]\!]$ from Sim and samples the honest parties' shares of $[\![r]\!]$ based on the secret and the corrupted parties' shares.
3. The trusted party distributes $[\![r]\!]$ to the parties.

Partially Authenticated Triples: Repeat the following m_P times:

1. The trusted party samples $a, b \in \mathbb{F}$ randomly and computes $c = a \cdot b$. Then the trusted party computes $\Delta \cdot a, \Delta \cdot b$.
2. The trusted party receives corrupted parties' shares of $[\![a]\!], [\![b]\!], [c]$ from Sim and samples the honest parties' shares of $[\![a]\!], [\![b]\!], [c]$ based on the secret and the corrupted parties' shares.
3. The trusted party distributes $[\![a]\!], [\![b]\!], [c]$ to the parties.

Authenticated Triples: Repeat the following m_A times:

1. The trusted party samples $a, b \in \mathbb{F}$ randomly and computes $c = a \cdot b$. Then the trusted party computes $\Delta \cdot a, \Delta \cdot b, \Delta \cdot c$.
2. The trusted party receives corrupted parties' shares of $[\![a]\!], [\![b]\!], [\![c]\!]$ from Sim and samples the honest parties' shares of $[\![a]\!], [\![b]\!], [\![c]\!]$ based on the secret and the corrupted parties' shares.
3. The trusted party distributes $[\![a]\!], [\![b]\!], [\![c]\!]$ to the parties.

Tensor Products: Repeat the following m_T times:

1. Run steps 1 and 2 from **Random Values** $2k$ times to create sharings $\{[\![a_i]\!], [\![b_i]\!]\}_{i=1,\ldots,k}$.
2. For each pair of $(i, j) \in \{1, \ldots, k\}^2$, the trusted party computes $c_{i,j} = a_i \cdot b_j$ and receives corrupted parties' shares of $[c_{i,j}]$ from Sim. Then, the trusted party samples the honest parties' shares of it based on the secret and the corrupted parties' shares.
3. The trusted party distributes $\{([\![a_i]\!], [\![b_i]\!])\}_{i=1,\ldots,k}, \{[c_{i,j}]\}_{i,j\in\{1,\ldots,k\}}$ to the parties.

Abort: If (abort, P) is received from Sim for an honest party P, the trusted party sends abort to P.

Figure 20: Functionality for the preprocessing phase.

G Preprocessing Protocol and Security Proof

Now, we begin to introduce our preprocessing protocol in the $\{\mathcal{F}^k_{\mathsf{patensor}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model. In particular, we only prepare partially authenticated tensor products. Random authenticated sharings and partially authenticated triples can be extracted from the tensor products. The authenticated triples are computed from partially authenticated triples using the technique of [GLM+24]. The protocol is given in Π_{prep} (Figure 21).

Protocol Π_{prep}

Public Parameters: Number m_R of random authenticated sharings, number m_P of partially authenticated triples, number m_A of authenticated triples, and number m_T of partially authenticated tensor products.

Init: The parties send $(\mathsf{PAtens}, \lceil (m_R + 2m_P + 6m_A)/2k \rceil + m_T)$ to $\mathcal{F}_{\mathsf{patensor}}^k$, receiving $[\Delta]$ and $\lceil (m_R + 2m_P + 6m_A)/2k \rceil + m_T$ partially authenticated triples $\{[\![a_i]\!], [\![b_i]\!]\}_{i=1}^k, \{[c_{i,j}]\}_{i,j \in \{1,\ldots,k\}}$. If abort is received, the party aborts the protocol.

Random Values: The parties extract $\{[\![a_i]\!], [\![b_i]\!]\}_{i=1}^k$ from the first $\lceil m_R/2k \rceil$ partially authenticated tensor products. Then, they already obtain m_R random authenticated sharings.

Partially Authenticated Triples: The parties extract triples $\{([\![a_i]\!], [\![b_i]\!], [c_{i,i}])\}_{i=1}^k$ from the first $\lceil (m_R + 2m_P)/2k \rceil$ partially authenticated tensor products. Except for those triples where $[\![a_i]\!], [\![b_i]\!]$ are consumed for **Random Values**, there remain at least m_P triples $([\![a]\!], [\![b]\!], [c])$ as the parties' outputs.

Authenticated Triples:

1. The parties extract triples $\{([\![a_i]\!], [\![b_i]\!], [c_{i,i}])\}_{i=1}^k$ from the first $\lceil (m_R + 2m_P + 6m_A)/2k \rceil$ partially authenticated tensor products. Except for those triples where $[\![a_i]\!], [\![b_i]\!]$ are consumed for **Random Values** and those triples consumed for **Partially Authenticated Triples**, there remain at least $3m_A$ triples $([\![a]\!], [\![b]\!], [c])$. The parties extract them, where each group of 3 triples $([\![a]\!], [\![b]\!], [c]), ([\![a']\!], [\![b']\!], [c']), ([\![\ell]\!], [\![\ell']\!], [c''])$ are put together to generate one authenticated triple.
2. For each group of triples, the parties compute their shares of $[\ell + c] = [\ell] + [c], [\ell' + c'] = [\ell'] + [c']$ and send them to P_{king}. P_{king} reconstructs $\ell + c, \ell' + c'$ and sends them to all the parties.
3. For each group of triples, each party P_i locally computes $[\![c]\!] = (\ell + c) - [\![\ell]\!], [\![c']\!] = (\ell' + c') - [\![\ell']\!]$.
4. The parties verify the triples as follows:
   (a) The parties call $\mathcal{F}_{\mathsf{Coin}}$ to get a random value $\alpha \in \mathbb{F}^*$.
   (b) For the triples $([\![a]\!], [\![b]\!], [\![c]\!]), ([\![a']\!], [\![b']\!], [\![c']\!])$ in the j-th group, the parties run Π_{Open} on $[\![e]\!] = [\![\alpha^j \cdot a + a']\!], [\![d]\!] = [\![b + b']\!]$.
   (c) The parties compute $[\![\tau_j]\!] = d \cdot e - e \cdot [\![b']\!] - d \cdot [\![a']\!] + [\![c']\!] - [\![\alpha^j \cdot c]\!]$.
   (d) After obtaining $[\![\tau_j]\!]$ for $j = 1, \ldots, m_A$, the parties run Π_{Open} on $[\![\tau]\!] = \sum_{\eta=1}^{m_A} [\![\tau_\eta]\!]$. If $\tau \neq 0$, the parties abort the protocol. Otherwise, they run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check all the opened values (i.e. d, e for each triple and τ).

Tensor Products: The parties extract the last m_T partially authenticate tensor products from the output of $\mathcal{F}_{\mathsf{patensor}}^k$ as their outputs.

Figure 21: Protocol for the preprocessing phase.

Lemma 1 *In the $\{\mathcal{F}_{\mathsf{patensor}}^k, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model, Π_{prep} securely realizes $\mathcal{F}_{\mathsf{prep}}$ with statistical security.*

We prove the security of Π_{prep} by constructing an ideal adversary Sim. Then we will show that the output in the ideal world is indistinguishable from that in the real world against the adversary $\mathcal{A}$ using hybrid arguments. Since the parties do not have inputs in $\mathcal{F}_{\mathsf{prep}}$, and the outputs of corrupted parties are fully determined by Sim in the ideal world (so there are no restrictions on honest parties' outputs other than one honest party's), we can assume that only one party P_h is honest in the protocol execution of Π_{prep}. If another party is honest, Sim just needs to honestly emulate him to run the protocol. Without loss of generality, we assume that P_{king} is corrupted. The ideal adversary Sim is provided below in Figure 22.

Simulator Sim

Init: Sim initializes $\mathsf{ErrorCheck} = \mathsf{MACCheck} = 0$. Sim emulates $\mathcal{F}_{\mathsf{patensor}}^k$ to receive $(\mathsf{PAtens}, \lceil (m_R + 2m_I + 6m_A)/2k \rceil + m_T)$ from the corrupted parties and corrupted parties' outputs of $\mathcal{F}_{\mathsf{patensor}}^k$ from Sim. Then, Sim emulates $\mathcal{F}_{\mathsf{patensor}}^k$ to distribute corrupted parties' outputs to them. If (abort, P_h) is received, Sim sends (abort, P_h) to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of P_h. After completing the simulation, Sim outputs the adversary's view.

Random Values: Sim follows the protocol to extract corrupted parties' shares of the random authenticated

sharings and then sends them to $\mathcal{F}_{\mathsf{prep}}$.

Partially Authenticated Triples: Sim follows the protocol to extract corrupted parties' shares of the partially authenticated triples and then sends them to $\mathcal{F}_{\mathsf{prep}}$.

Authenticated Triples:

1. Sim follows the protocol to extract corrupted parties' shares of triples $([\![a]\!], [\![b]\!], [c]), ([\![a']\!], [\![b']\!], [c']), ([\![\ell]\!], [\![\ell']\!], [c''])$ in each group.
2. For each group of triples, Sim samples two random field elements as P_h's shares of $[\ell + c], [\ell' + c']$ and sends them to P_{king} on behalf of P_h. Sim then follows the protocol to compute corrupted parties' shares of $[\ell + c], [\ell' + c']$ and reconstructs $\ell + c, \ell' + c'$ from all parties' shares.
3. For the j-th group of triples, Sim receives $\overline{\ell + c}, \overline{\ell' + c'}$ from P_{king}. Then, Sim computes $\delta_j = \overline{\ell + c} = (\ell + c)$ and $\delta'_j = \overline{\ell' + c'} = (\ell' + c')$. If $\delta_j \neq 0$, Sim sets $\mathsf{ErrorCheck} = 1$.
4. Sim receives RandCoin from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send α to them. If Sim receives abort from $\mathcal{A}$, he sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.
5. For each group of triples, Sim samples two random field elements as P_h's shares of $[e], [d]$ and sends them to P_{king} on behalf of P_h. Sim then follows the protocol to compute corrupted parties' shares of $[e], [d]$ and reconstructs e, d from all parties' shares.
6. For each group of triples, Sim receives $\overline{d}, \overline{e}$ from P_{king}. Then, if $\overline{d} \neq d$ or $\overline{e} \neq e$, Sim sets $\mathsf{MACCheck} = 1$.
7. Sim computes $\tilde{\tau} = \sum_{j=1}^{m_A} (\delta'_j - \alpha^j \delta_j)$. If $\mathsf{ErrorCheck} = 1$ but $\tilde{\tau} = 0$, Sim aborts the simulation.
8. If $\mathsf{ErrorCheck} = 1$ or $\mathsf{MACheck} = 1$, Sim samples P_h's shares of $[a], [b], [a'], [b']$ in each group based on the corresponding shares of $[e], [d]$. Then, Sim follows the protocol to compute P_h's share of $[\tau]$. Otherwise, Sim follows the protocol to compute corrupted parties' shares of $[\tau]$ and computes P_h's share based on their shares and the secret $\tau = 0$.
9. Sim sends P_h's share of $[\tau]$ to P_{king} and then receives τ from P_{king}. If $\tau \neq \tilde{\tau}$, Sim sets $\mathsf{MACCheck} = 1$. If $\tau \neq 0$, Sim aborts the protocol on behalf of P_h. After the simulation completes, Sim outputs the adversary's view.
10. Sim simulates the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as follows:
    (a) Sim receives RandCoin from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send γ to them. If Sim receives abort from $\mathcal{A}$, he sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.
    (b) Sim follows the protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to compute corrupted parties' shares of $[\sigma]$.
    (c) Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, P_i, \sigma_i, \mathsf{mid}_{\sigma_i})$ from each corrupted party P_i.
    (d)
    - If $\mathsf{MACCheck} = 0$, Sim computes the P_h's share of $[\sigma]$ based on the corrupted parties' shares (computed by Sim honestly, not the received shares by $\mathcal{F}_{\mathsf{Commit}}$) and $\sum_{i=1}^{n} \sigma_i = 0$.
    - If $\mathsf{MACCheck} = 1$, Sim sends abort to $\mathcal{F}_{\mathsf{prep}}$ and receives Δ from $\mathcal{F}_{\mathsf{prep}}$. For each value A_i opened in Π_{Open} protocol:
      * Sim computes $\Delta \cdot A_i$ and computes the honest party's share of $[\Delta \cdot A_i]$ based on the corrupted parties' shares and $\Delta \cdot A_i$. Each A_i is reconstructed based on all the parties' shares, where the corrupted parties' shares are computed by following the protocol. In particular, the opened value τ' is set to $\tilde{\tau'}$.
      * Sim follows the protocol to compute the P_h's share σ_h of $[\sigma]$.

    (e) For each party corrupted P_i, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties. For the honest party P_h, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_h, \mathsf{mid}_{\sigma_h})$ to all the corrupted parties.
    (f) For each party P_i, Sim receives $(\mathsf{open}, P_i, \mathsf{mid}_{\sigma_i})$ from the corrupted parties and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\sigma_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties.
    (g) Sim follows the protocol to check whether $\sum_{i=1}^{n} \sigma_i = 0$ holds. If not, Sim aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.
    (h) If $\mathsf{MACCheck} = 1$, Sim aborts the simulation.

11. Sim follows the protocol to compute corrupted parties' shares of the authenticated triples and sends them to $\mathcal{F}_{\mathsf{prep}}$.

Tensor Products: Sim follows the protocol to extract corrupted parties' shares of the partially authenticated tensor products and then sends them to $\mathcal{F}_{\mathsf{prep}}$.

Figure 22: The simulator for Π_{prep}.

We construct the following hybrids:

$\mathbf{Hyb}_0$: In this hybrid, Sim runs the protocol honestly on behalf of the honest parties and communicates with $\mathcal{A}$. This corresponds to the real-world scenario.

$\mathbf{Hyb}_1$: In this hybrid, Sim additionally initializes $\mathsf{ErrorCheck} = \mathsf{ErrorCheck} = \mathsf{MACCheck} = 0$ at the beginning. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ have the same output distribution.

$\mathbf{Hyb}_2$: In this hybrid, Sim does not generate the honest party P_h's outputs from $\mathcal{F}^k_{\mathsf{patensor}}$ at the beginning. Instead, Sim generates them whenever they are used. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

$\mathbf{Hyb}_3$: In this hybrid, during the preparation of authenticated triples, for each group of triples, Sim does not follow the protocol to generate the honest party P_h's shares of $[\ell], [\ell']$ and compute P_h's shares of $[\ell + c], [\ell' + c']$. Instead, Sim samples 2 random field elements as them and then computes P_h's shares of $[\ell], [\ell']$ based on his shares of $[\ell + c], [\ell' + c']$ and $[c], [c']$. Since P_h's shares of $[\ell], [\ell']$ are uniformly random, so are his shares of $[\ell + c], [\ell' + c']$. Therefore, we only change the order of generating P_h's shares of $[\ell], [\ell']$ and $[\ell + c], [\ell' + c']$ without changing their distributions. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same output distribution.

$\mathbf{Hyb}_4$: In this hybrid, during the preparation of authenticated triples, for the j-th group of triples, after receiving $\overline{\ell + c}, \overline{\ell' + c'}$ from P_{king}, Sim additionally computes $\delta_j = \overline{\ell + c} = (\ell + c)$ and $\delta'_j = \overline{\ell' + c'} = (\ell' + c')$. If $\delta_j \neq 0$, Sim sets $\mathsf{ErrorCheck} = 1$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ have the same output distribution.

$\mathbf{Hyb}_5$: In this hybrid, during the verification of authenticated triples, for the j-th group of triples, Sim does not first generate the honest party P_h's shares of $[a'], [b']$ and then compute P_h's shares of $[e], [d]$. Instead, Sim samples 2 random field elements as them and then computes P_h's shares of $[a'], [b']$ based on his shares of $[e] = [\alpha^j \cdot a + a'], [d] = [b + b']$ and $[a], [b]$. Since P_h's shares of $[a'], [b']$ are uniformly random, so are his shares of $[e], [d]$. Therefore, we only change the order of generating P_h's shares of $[a'], [b']$ and $[e], [d]$ without changing their distributions. Thus, $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same output distribution.

$\mathbf{Hyb}_6$: In this hybrid, during the verification of authenticated triples, for the j-th group of triples, after receiving $\overline{d}, \overline{e}$ from P_{king}, Sim additionally checks whether $\overline{d} = d, \overline{e} = e$. If not, Sim sets $\mathsf{MACCheck} = 1$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ have the same output distribution.

$\mathbf{Hyb}_7$: In this hybrid, during the verification of authenticated triples, Sim additionally computes $\tilde{\tau} = \sum_{j=1}^{m_A} (\delta'_j - \alpha^j \delta_j)$. If $\mathsf{ErrorCheck} = 1$ but $\tilde{\tau} = 0$, Sim aborts the simulation.

Suppose that for a $j \in \{1, \ldots, m_A\}$, $\delta_j \neq 0$. Then,

$$\tilde{\tau} = \sum_{j=1}^{m_A} (\delta'_j - \alpha^j \delta_j)$$

is a non-zero polynomial in α. Note that α is sent to the adversary after the errors are determined, so the distribution only changes when α happens to be a root for the non-zero polynomial. Since the polynomial is of degree $m_A = \mathsf{poly}(\kappa)$, there are at most $\mathsf{poly}(\kappa)$ roots. Therefore, the probability that α is one of them is negligible. Thus, the distributions of $\mathbf{Hyb}_7$ and $\mathbf{Hyb}_6$ ar statistically close.

$\mathbf{Hyb}_8$: In this hybrid, during the verification of authenticated triples, if $\mathsf{ErrorCheck} = \mathsf{MACCheck} = 0$, Sim does not follow the protocol to compute P_h's shares of $[\tau]$. Sim follows the protocol to compute corrupted parties' shares of $[\tau]$ and then computes P_h's share of $[\tau]$ based on the secret $\tau = 0$ and corrupted parties' shares. Note that in this case, the corrupted parties generate valid triples and honestly compute $[\tau]$.

Therefore, $\tau = 0$ must hold, so we only change the way of computing P_h's share of $[\tau]$ without changing its distribution. Thus, $\mathbf{Hyb}_8$ and $\mathbf{Hyb}_7$ have the same output distribution.

$\mathbf{Hyb}_9$: In this hybrid, during the verification of authenticated triples, if the opened secret $\tilde{\tau}$ of $[\tau]$ by P_{king} does not match the previously computed τ, Sim additionally sets $\mathsf{MACCheck} = 1$. Note that τ is computed to be the secret of $[\tau]$ shared by all parties, where all d, e are assumed to be opened correctly, so $\mathsf{MACCheck} = 1$ only holds when at least one of d, e for a group of triples or τ is not opened correctly. This does not affect the output distribution. Thus, $\mathbf{Hyb}_9$ and $\mathbf{Hyb}_8$ have the same output distribution.

$\mathbf{Hyb}_{10}$: In this hybrid, during the verification of authenticated triples, while simulating $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, if $\mathsf{MACCheck} = 0$, Sim does not follow the protocol to compute P_h's share of $[\sigma]$. Instead, Sim computes the P_h's share of $[\sigma]$ based on the corrupted parties' shares and $\sum_{i=1}^{n} \sigma_i = 0$. Note that when $\mathsf{MACCheck} = 0$, all the opened values are opened correctly via Π_{Open}. In this case, it must hold that $[\sigma] = [0]$, so we only change the way of generating P_h's share of $[\sigma]$ without changing its distribution. Thus, $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ have the same output distribution.

$\mathbf{Hyb}_{11}$: In this hybrid, during the verification of authenticated triples, while simulating $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, if $\mathsf{MACCheck} = 1$, Sim doesn't directly compute the honest party P_h's share of $[\Delta \cdot A_i]$ for each opened value A_i. Instead, Sim first computes each $\Delta \cdot A_i$, and then Sim computes the honest party's share of $[\Delta \cdot A_i]$ based on the corrupted parties' shares and $\Delta \cdot A_i$. We just change the way the honest party's share of each $[\Delta \cdot A_i]$ is generated without changing its distribution. Thus, $\mathbf{Hyb}_{11}$ and $\mathbf{Hyb}_{10}$ have the same output distribution.

$\mathbf{Hyb}_{12}$: In this hybrid, during the verification of authenticated triples, while simulating $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, after following the protocol to check whether $\sum_{i=1}^{n} \sigma_i = 0$, Sim aborts the simulation if $\mathsf{MACCheck} = 1$. The only difference between $\mathbf{Hyb}_{12}$ and $\mathbf{Hyb}_{11}$ is that the corrupted parties may not open all the values of SPDZ sharing checked in this execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ correctly, but the verification of MACs may pass.

We suppose that $A_1, \ldots, A_m$ are opened with additive errors $\delta^{(1)}, \ldots, \delta^{(m)}$, and the sum of all the committed shares $[\sigma]$ of corrupted parties is with an additive error δ_σ. Then

$$\begin{aligned}\sigma =& \sum_{i\in[1,n],i\neq h}\left(\sum_{\alpha=1}^{m}\chi^\alpha\cdot(\Delta\cdot A_\alpha)^{(i)} - \Delta_i\cdot\sum_{\alpha=1}^{m}\chi^\alpha\cdot A_\alpha\right) + \delta_\sigma\\ &+\left(\sum_{\alpha=1}^{m}\chi^\alpha\cdot(\Delta\cdot A_\alpha)^{(h)} - \Delta_h\cdot\sum_{\alpha=1}^{m}\chi^\alpha\cdot(A_\alpha+\delta^{(\alpha)})\right)\\ =&\delta_\sigma + \Delta_h\cdot\sum_{\alpha=1}^{m}\chi^\alpha\cdot\delta^{(\alpha)}.\end{aligned}$$

Here $(\Delta\cdot A_\alpha)^{(i)}$ is used to denote P_i's share of $[\Delta\cdot A_\alpha]$. Since Δ_h is not used in computing any message sent to the corrupted parties at this moment, $\Delta_h\cdot\sum_{\alpha=1}^{m}\chi^\alpha\cdot\delta^{(\alpha)}$ is a random element in $\mathbb{F}$ as long as $\sum_{\alpha=1}^{m}\chi^\alpha\cdot\delta^{(\alpha)}$ is non-zero. When $(\delta^{(1)},\ldots,\delta^{(m)})$ is a nonzero vector, since a non-zero degree-m polynomial only has at most m zeros, so $\sum_{\alpha=1}^{m}\chi^\alpha\cdot\delta^{(\alpha)} = 0$ only happens with at most $m/|\mathbb{F}|$ probability. Since $m = \mathsf{poly}(n,\kappa)$, the probability is negligible. On the other hand, in the case where $\sum_{\alpha=1}^{m}\chi^\alpha\cdot\delta^{(\alpha)} = 0$, σ is opened to a random element in $\mathbb{F}$, which equals 0 with $1/|\mathbb{F}|$ probability, which is also negligible.

Thus, the distributions of $\mathbf{Hyb}_{12}$ and $\mathbf{Hyb}_{11}$ are statistically close.

$\mathbf{Hyb}_{13}$: In this hybrid, P_h does not follow the protocol to compute his output. Instead, Sim sends corrupted parties' outputs to $\mathcal{F}_{\mathsf{prep}}$, and honest parties obtain their outputs from $\mathcal{F}^k_{\mathsf{uatensor}}$. We only change how honest parties' outputs are generated, but the computational process of honest parties' outputs by Sim and by $\mathcal{F}_{\mathsf{prep}}$ is the same. Therefore, the output distribution remains unchanged. Thus, $\mathbf{Hyb}_{13}$ and $\mathbf{Hyb}_{12}$ have the same distribution.

Note that $\mathbf{Hyb}_{13}$ is the ideal-world scenario. Thus, Π_{prep} computes $\mathcal{F}_{\mathsf{prep}}$ with computational security.

H The Protocol Description of Π_{main}

Our main protocol Π_{main} is presented in Figure 23 below.

Protocol Π_{main}

1. **Preprocessing.** Let m_I, m_O be the number of input and output wires in C respectively, and let $\ell = \lceil |C|/k \rceil$. The parties invoke $\mathcal{F}_{\mathsf{prep}}$ with inputs $(\mathsf{Prep}, m_R, m_I + m_O, m_A, m_T)$ to $\mathcal{F}_{\mathsf{prep}}$ and receive the preprocessing data from $\mathcal{F}_{\mathsf{prep}}$, where $m_R = m_O + 2\ell(\lambda - 1) \cdot \lceil \log_\lambda k \rceil + \ell - 1$, $m_A = \ell$, and $m_T = \lceil |C|/k \rceil$.
2. **Evaluation.** The parties run Π_{eval} (Figure 7).
3. **Verification.** The parties run Π_{ver} (Figure 8).
4. **Output Reconstruction.** For each output wire with value v of C', the parties run Π_{Open} on $[\![v]\!]$. Then, the parties run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on all the opened output values and send each output wire value to the designated output party. Each party then checks whether the received outputs are all the same. If not, the party aborts the protocol. Otherwise, he locally computes his output of C from his output of C' and the masks he inputs to C'.

Figure 23: The main protocol.

Theorem 1 *In the $\{\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model, Π_{main} is a secure multiparty computation protocol against any adversary $\mathcal{A}$ that corrupts up to $t = n - 1$ parties.*

We prove the security of Π_{main} by constructing an ideal adversary Sim. Then we will show that the output in the ideal world is indistinguishable from that in the real world against the adversary $\mathcal{A}$ using hybrid arguments. Without loss of generality, we assume that P_{king} is corrupted. Let P_h be an honest party. The ideal adversary Sim is provided below in Figure 24.

Simulator Sim

At the beginning of the simulation, Sim sets $\mathsf{CompCheck} = \mathsf{MACCheck} = 0$. Then:

1. **Preprocessing.** Sim faithfully emulates $\mathcal{F}_{\mathsf{prep}}$ to receive the requests from corrupted parties and the corrupted parties' outputs from $\mathcal{A}$. Then, Sim sends the corrupted parties' outputs to them. If (abort, P) is received for an honest party P, then Sim randomly samples $\Delta \in \mathbb{F}$, sends it to $\mathcal{A}$, and aborts the protocol on behalf of P. After the simulation completes, Sim outputs the adversary's view.
2. **Evaluation.** Sim runs $\mathsf{Sim}_{\mathsf{eval}}$ (Figure 25).
3. **Verification.** Sim runs $\mathsf{Sim}_{\mathsf{ver}}$ (Figure 26).
4. **Output Reconstruction.**
   - For each output wire value v of C' attached to an honest party, Sim randomly samples a field element as P_h's share of $[v]$ and follows the protocol to compute other parties' shares of $[v]$.
   - For each output wire value v of C' attached to a corrupted party, Sim receives the corresponding output wire value of C from $\mathcal{F}$, and then computes v from the corrupted party's inputs of C' and his outputs of C. Sim then follows the protocol to compute other parties' shares of $[v]$ and computes P_h's share of it based on the secret and other parties' shares.

   After that, Sim sends the honest parties' shares of $[v]$ to P_{king} on their behalf and receives $\overline{v}$ from P_{king}. Sim then checks whether all parties' shares of $[v]$ reconstruct to $\overline{v}$. If not, Sim sets $\mathsf{MACCheck} = 1$. After opening all the outputs of C', Sim simulates the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as follows:

   (a) Sim receives $\mathsf{RandCoin}$ from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send the seed to them. If Sim receives abort from $\mathcal{A}$, he sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.

   (b) For each party $P_i \neq P_h$, Sim follows the protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to compute P_i's share of $[\sigma]$.

   (c) Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, P_i, \sigma_i, \mathsf{mid}_{\sigma_i})$ from each corrupted party P_i.

   (d)
   - If $\mathsf{MACCheck} = 0$, Sim computes the P_h's share of $[\sigma]$ based on the other parties' shares (computed by Sim honestly, not the received shares by $\mathcal{F}_{\mathsf{Commit}}$) and $\sum_{i=1}^{n} \sigma_i = 0$, where the parties' shares are computed using the regarded messages from corrupted parties.
   - If $\mathsf{MACCheck} = 1$, Sim randomly samples $\Delta \in \mathbb{F}$. For each value A_i opened in Π_{Open} protocol:

* Sim computes $\Delta \cdot A_i$ and computes the honest party's share of $[\Delta \cdot A_i]$ based on the corrupted parties' shares and $\Delta \cdot A_i$. Each A_i is reconstructed based on all the parties' shares, where the corrupted parties' shares are computed by following the protocol.
* Sim follows the protocol to compute the P_h's share σ_h of $[\sigma]$.

(e) For each party corrupted P_i, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties. For each honest party P_j, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_j, \mathsf{mid}_{\sigma_j})$ to all the corrupted parties, where each σ_j with $j \neq h$ is computed using the actual messages received from corrupted parties.

(f) For each party P_i, Sim receives $(\mathsf{open}, P_i, \mathsf{mid}_{\sigma_i})$ from the corrupted parties and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\sigma_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties.

(g) Sim follows the protocol to check whether $\sum_{i=1}^{n} \sigma_i = 0$ holds. If not, Sim aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.

(h) If $\mathsf{MACCheck} = 1$, Sim aborts the simulation.

Finally, Sim outputs the view of the adversary.

Figure 24: The simulator for the main protocol.

Simulator $\mathsf{Sim}_{\mathsf{eval}}$

Sim follows the protocol to obtain circuit C'. Then:

1. **Authenticating Inputs.** For each input wire of C' attached to a party P_i:
   - If P_i is honest, Sim receives corrupted parties' shares of $([a], [b], [c])$ from the consumed partially authenticated triple on behalf of P_i. Then, Sim checks whether the sum of corrupted parties' shares of $[a]$ (resp. $[b], [c]$) received from them is the same as the sum of corrupted parties' shares of $[a]$ (resp. $[b], [c]$) sent to them by Sim while emulating $\mathcal{F}_{\mathsf{prep}}$. If not, Sim aborts the protocol on behalf of P_i. After completing the simulation, Sim outputs the adversary's view. Otherwise, Sim samples a random element as $x + a$ and sends it to all the corrupted parties.
   - If P_i is corrupted, Sim randomly samples 2 field elements as each honest party's share of $[a], [b]$ from the consumed input mask sharings and then computes a, b, c. Then, Sim samples the honest parties' shares of $[c]$ based on the secret and corrupted parties' shares and sends them to P_i on behalf of the honest parties. Sim then receives $x + a$ from P_i on behalf of P_h and computes $x = (x + a) - a$ as P_i's input on this wire. Sim regards that all parties receive the same message $x + a$ as P_h.

   After obtaining all the corrupted parties' inputs to C, Sim sends them to $\mathcal{F}$ and obtains the outputs of corrupted parties for the circuit C.
2. **Evaluating the Circuit.** For each honest party $P_i \neq P_h$, Sim faithfully samples the remaining outputs for P_i from $\mathcal{F}_{\mathsf{prep}}$. During the evaluation, the transcripts between these parties and the corrupted parties are all computed honestly from the actual transcripts they received. For each input wire with input x, Sim follows the protocol to compute each $P_i \neq P_h$'s share of $[\![x]\!]$. Sim then follows the protocol to group the multiplication gates. For each gate of C':
   - If the gate is an addition gate, Sim follows the protocol to evaluate each $P_i \neq P_h$'s share of the additive sharing for the output wire value of this gate.
   - If the gate is a multiplication gate with input wires $x_\alpha^{(j)}, y_\alpha^{(j)}$:
     (a) Sim samples P_h's shares of $[x_\alpha^{(j)} + a_\alpha], [y_\alpha^{(j)} + b_\alpha]$ randomly.
     (b) Sim sends the honest parties' shares of $[x_\alpha^{(j)} + a_\alpha], [y_\alpha^{(j)} + b_\alpha]$ to P_{king} on behalf of the honest parties (the shares of honest parties except P_h are computed by following the protocol using actual transcripts they receive from the corrupted parties). Sim then reconstructs $x_\alpha^{(j)} + a_\alpha, y_\alpha^{(j)} + b_\alpha$ from all parties' shares (the shares of all parties except P_h are computed using the regarded messages, i.e., when a corrupted party needs to send a message to all parties, we regard that all honest parties receive the message for P_h. Same below.).
     (c) Sim receives $\overline{x_\alpha^{(j)} + a_\alpha}, \overline{y_\alpha^{(j)} + b_\alpha}$ from P_{king} on behalf of P_h. Sim then regards that all the honest parties receive $\overline{x_\alpha^{(j)} + a_\alpha}, \overline{y_\alpha^{(j)} + b_\alpha}$ from P_{king} as $x_\alpha^{(j)} + a_\alpha, y_\alpha^{(j)} + b_\alpha$. Then, Sim additionally follows

the protocol to compute each $P_i \neq P_h$'s share of $[\![x_\alpha^{(j)}]\!], [\![y_\alpha^{(j)}]\!], [z_\alpha^{(j)}]$. The honest parties' shares among them are only used for doing reconstructions (e.g., in Step (b)), while the transcripts are still computed with the actual values they receive from corrupted parties.

(d) * If $x_\alpha^{(j)}$ (resp. $y_\alpha^{(j)}$) can be computed directly from input wires of C', Sim sets $\mathsf{MACCheck} = 1$ if $\overline{x_\alpha^{(j)} + a_\alpha} \neq x_\alpha^{(j)} + a_\alpha$ (resp. $\overline{y_\alpha^{(j)} + b_\alpha} \neq y_\alpha^{(j)} + b_\alpha$).

* Otherwise, Sim computes $\delta_{x_\alpha^{(j)}} = \overline{x_\alpha^{(j)} + a_\alpha} - (x_\alpha^{(j)} + a_\alpha)$ (resp. $\delta_{y_\alpha^{(j)}} = \overline{y_\alpha^{(j)} + b_\alpha} - (y_\alpha^{(j)} + b_\alpha)$). If $\overline{x_\alpha^{(j)} + a_\alpha} \neq x_\alpha^{(j)} + a_\alpha$ (resp. $\overline{y_\alpha^{(j)} + b_\alpha} \neq y_\alpha^{(j)} + b_\alpha$), Sim sets $\mathsf{CompCheck} = 1$.

Finally, for each output wire with value v of C':

(a) Sim samples P_h's share of $[r + v]$ randomly.

(b) Sim sends the honest parties' shares of $[r + v]$ to P_{king} on behalf of the honest parties. Sim then reconstructs $r + v$ from all parties' shares.

(c) Sim receives $\overline{r + v}$ from P_{king} on behalf of P_h. Sim then regards that all the honest parties receive $\overline{r + v}$ from P_{king} as $r + v$. Then, Sim follows the protocol to compute each $P_i \neq P_h$'s share of $[\![v]\!]$.

(d) – If v an be computed directly from input wires of C', Sim sets $\mathsf{MACCheck} = 1$ if $\overline{r + v} \neq r + v$.

– Otherwise, Sim computes $\delta_v = \overline{r + v} - (r + v)$. If $\overline{r + v} \neq r + v$, Sim sets $\mathsf{CompCheck} = 1$.

3. **MAC Check.** Sim simulates the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as follows:

(a) Sim receives $\mathsf{RandCoin}$ from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send the seed to them. If Sim receives abort from $\mathcal{A}$, he sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.

(b) For each party $P_i \neq P_h$, Sim follows the protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to compute P_i's share of $[\sigma]$.

(c) Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, P_i, \sigma_i, \mathsf{mid}_{\sigma_i})$ from each corrupted party P_i.

(d) – If $\mathsf{MACCheck} = 0$, Sim computes the P_h's share of $[\sigma]$ based on the other parties' shares (computed by Sim honestly, not the received shares by $\mathcal{F}_{\mathsf{Commit}}$) and $\sum_{i=1}^n \sigma_i = 0$, where the parties' shares are computed using the regarded messages from corrupted parties.

– If $\mathsf{MACCheck} = 1$, Sim randomly samples $\Delta \in \mathbb{F}$. For each value A_i opened in Π_{Open} protocol:

* Sim computes $\Delta \cdot A_i$ and computes the honest party's share of $[\Delta \cdot A_i]$ based on the corrupted parties' shares and $\Delta \cdot A_i$. Each A_i is reconstructed based on all the parties' shares, where the corrupted parties' shares are computed by following the protocol.

* Sim follows the protocol to compute the P_h's share σ_h of $[\sigma]$.

(e) For each party corrupted P_i, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties. For each honest party P_j, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_j, \mathsf{mid}_{\sigma_j})$ to all the corrupted parties, where each σ_j with $j \neq h$ is computed using the actual messages received from corrupted parties.

(f) For each party P_i, Sim receives $(\mathsf{open}, P_i, \mathsf{mid}_{\sigma_i})$ from the corrupted parties and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\sigma_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties.

(g) Sim follows the protocol to check whether $\sum_{i=1}^n \sigma_i = 0$ holds. If not, Sim aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.

(h) If $\mathsf{MACCheck} = 1$, Sim aborts the simulation.

Figure 25: The simulator for the evaluation phase.

Simulator $\mathsf{Sim}_{\mathsf{ver}}$

1. **Reducing to Inner-Product Check.**

(a) Sim receives $\mathsf{RandCoin}$ from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send s to them. If Sim receives abort from $\mathcal{A}$, he sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.

(b) For each party $P_i \neq P_h$, Sim follows the protocol to compute each P_i's shares of $[\![\boldsymbol{x}]\!], [\![\boldsymbol{y}]\!], [\![z]\!]$. Moreover, Sim computes $\delta_z = \sum_{j=1}^M s^j \delta_{z_j}$, where δ_{z_j} is the additive error attached to the value z_j computed during

the evaluation phase.

(c) If $\mathsf{CompCheck} = 1$ but $\delta_z = 0$, Sim aborts the simulation.

(d) For $[\![\boldsymbol{x}]\!] = \left(\sum_{i=1}^{m} s^i \cdot \alpha_i^{(1)} \cdot [\![x_1]\!], \ldots, \sum_{i=1}^{m} s^i \cdot \alpha_i^{(|C|)} \cdot [\![x_{|C|}]\!] \right)$, if there exists $j \in \{1, \ldots, |C|\}$ such that the coefficient $\theta_j = \sum_{i=1}^{M} s^i \cdot \alpha_i^{(j)} = 0$, Sim aborts the simulation.

2. **Separate into Blocks.**

(a) For each $j = 1, \ldots, \ell - 1$:

i. Sim samples P_h's share of $[r^{(j)} + z^{(j)}]$ randomly.

ii. Sim sends the honest parties' shares of $[r^{(j)} + z^{(j)}]$ to P_{king} on behalf of the honest parties. Sim then reconstructs $r^{(j)} + z^{(j)}$ from all parties' shares.

iii. Sim receives $\overline{r^{(j)} + z^{(j)}}$ from P_{king} on behalf of P_h. Sim then regards that all the honest parties receive $\overline{r^{(j)} + z^{(j)}}$ from P_{king} as $r^{(j)} + z^{(j)}$. Then, Sim follows the protocol to compute each $P_i \neq P_h$'s share of $[\![z^{(j)}]\!]$.

iv. Sim computes $\delta_{z^{(j)}} = \overline{r^{(j)} + z^{(j)}} - (r^{(j)} + z^{(j)})$.

Sim then follows the protocol to compute each $P_i \neq P_h$'s share of $[\![z^{(\ell)}]\!]$ and then computes $\delta_{z^{(\ell)}} = \delta_z - \sum_{j=1}^{\ell-1} \delta_{z^{(j)}}$.

3. **Recursive Check.** For each group of multiplication gates (with index j), Sim follows the protocol to do initializations on behalf of each $P_i \neq P_h$ and compute P_i's share of $[\boldsymbol{u} \otimes \boldsymbol{v}]$. Sim additionally sets $\delta_w = \delta_{z^{(j)}}$. For each reduction in vector length:

(a) For each $i = 1, \ldots, \lambda - 1$:

i. Sim samples P_h's share of $[r_i + w_i]$ randomly.

ii. Sim sends the honest parties' shares of $[r_i + w_i]$ to P_{king} on behalf of the honest parties. Sim then reconstructs $r_i + w_i$ from all parties' shares.

iii. Sim receives $\overline{r_i + w_i}$ from P_{king} on behalf of P_h. Sim then regards that all the honest parties receive $\overline{r_i + w_i}$ from P_{king} as $r_i + w_i$. Then, Sim follows the protocol to compute each $P_i \neq P_h$'s share of $[\![w_i]\!]$.

iv. Sim computes $\delta_{w_i} = \overline{r_i + w_i} - (r_i + w_i)$.

Finally, Sim follows the protocol to compute each $P_i \neq P_h$'s share of $[\![w_\lambda]\!]$ and then computes $\delta_{w_\lambda} = \delta_w - \sum_{i=1}^{\lambda-1} \delta_{w_i}$.

(b) For each $i = \lambda + 1, \ldots, 2\lambda - 1$:

i. Sim samples P_h's share of $[r_i + h(i)]$ randomly.

ii. Sim sends the honest parties' shares of $[r_i + h(i)]$ to P_{king} on behalf of the honest parties. Sim then reconstructs $r_i + h(i)$ from all parties' shares.

iii. Sim receives $\overline{r_i + h(i)}$ from P_{king} on behalf of P_h. Sim then regards that all the honest parties receive $\overline{r_i + h(i)}$ from P_{king} as $r_i + h(i)$. Then, Sim follows the protocol to compute each $P_i \neq P_h$'s share of $[\![h(i)]\!]$.

iv. Sim computes $\delta_{h(i)} = \overline{r_i + h(i)} - (r_i + h(i))$.

(c) Sim receives $\mathsf{RandCoin}$ from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send r to them. If Sim receives abort from $\mathcal{A}$, he sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.

(d) For the linear function L' such that $h(r) = L'(h(1), \ldots, h(2\lambda - 1))$, Sim computes $\delta_{h(r)} = L'(\delta_{w_1}, \ldots, \delta_{w_\lambda}, \delta_{h(\lambda+1)}, \ldots, \delta_{h(2\lambda-1)})$.

(e) If $\delta_w \neq 0$ but $\delta_{h(r)} = 0$, Sim aborts the simulation. Otherwise, Sim updates $\delta_w = \delta_{h(x)}$.

(f) Sim follows the protocol to update each $P_i \neq P_h$'s shares of $[\![\boldsymbol{u}]\!], [\![\boldsymbol{v}]\!], [\![w]\!], [\boldsymbol{u} \otimes \boldsymbol{v}]$.

4. **Verify the Final Triple.** After the parties obtain the final triple $([\![u]\!], [\![v]\!], [\![w]\!])$ with error δ_w for each group of multiplication gates:

(a) Sim samples P_h's shares of $[u + a], [v + b]$ randomly.

(b) Sim sends the honest parties' shares of $[u + a], [v + b]$ to P_{king} on behalf of the honest parties (the shares of honest parties except P_h are computed by following the protocol using actual transcripts they receive from the corrupted parties). Sim then reconstructs $u + a, v + b$ from all parties' shares.

(c) Sim receives $\overline{u+a}, \overline{v+b}$ from P_{king} on behalf of P_h. Sim then regards that all the honest parties receive $\overline{u+a}, \overline{v+b}$ from P_{king} as $u+a, v+b$.

(d) Sim sets $\mathsf{MACCheck} = 1$ if $\overline{u+a} \neq u+a$ or $\overline{v+b} \neq v+b$.

(e)
- If $\mathsf{MACCheck} = 1$ or $\delta_w \neq 0$, Sim samples P_h's shares of $[u], [a], [v], [b]$ based on his shares of $[u+a], [v+b]$. Then, Sim reconstructs u, v, a, b, computes $w = uv + \delta_w, c = ab$, and computes P_h's shares of $[w], [c]$ based on corrupted parties' shares and the secrets. Sim then follows the protocol to compute P_h's share of $[\tau]$ and reconstruct τ (with the regarded messages).
- Otherwise, Sim honestly computes the shares of $[\tau]$ for each $P_i \neq P_h$ with the regarded messages and then computes P_h's share based on the secret $\tau = 0$ and other parties' shares.

(f) Sim sends the honest parties' shares of $[\tau]$ to P_{king} on behalf of the honest parties and receives $\overline{\tau}$ from P_{king}. Sim follows the protocol to check whether $\overline{\tau} = 0$ on behalf of each honest party. If not, Sim aborts the protocol and outputs the adversary's view after completing the simulation. Furthermore, if $\overline{\tau} \neq \tau$, Sim sets $\mathsf{MACCheck} = 1$.

(g) Sim simulates the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as follows:

i. Sim receives $\mathsf{RandCoin}$ from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send the seed to them. If Sim receives abort from $\mathcal{A}$, he sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.

ii. For each party $P_i \neq P_h$, Sim follows the protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to compute P_i's share of $[\sigma]$.

iii. Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, P_i, \sigma_i, \mathsf{mid}_{\sigma_i})$ from each corrupted party P_i.

iv.
- If $\mathsf{MACCheck} = 0$, Sim computes the P_h's share of $[\sigma]$ based on the other parties' shares (computed by Sim honestly, not the received shares by $\mathcal{F}_{\mathsf{Commit}}$) and $\sum_{i=1}^{n} \sigma_i = 0$, where the parties' shares are computed using the regarded messages from corrupted parties.
- If $\mathsf{MACCheck} = 1$, Sim randomly samples $\Delta \in \mathbb{F}$. For each value A_i opened in Π_{Open} protocol:
  - Sim computes $\Delta \cdot A_i$ and computes the honest party's share of $[\Delta \cdot A_i]$ based on the corrupted parties' shares and $\Delta \cdot A_i$. Each A_i is reconstructed based on all the parties' shares, where the corrupted parties' shares are computed by following the protocol.
  - Sim follows the protocol to compute the P_h's share σ_h of $[\sigma]$.

v. For each party corrupted P_i, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties. For each honest party P_j, Sim emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{committed}, P_j, \mathsf{mid}_{\sigma_j})$ to all the corrupted parties, where each σ_j with $j \neq h$ is computed using the actual messages received from corrupted parties.

vi. For each party P_i, Sim receives $(\mathsf{open}, P_i, \mathsf{mid}_{\sigma_i})$ from the corrupted parties and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\sigma_i, \mathsf{mid}_{\sigma_i})$ to all the corrupted parties.

vii. Sim follows the protocol to check whether $\sum_{i=1}^{n} \sigma_i = 0$ holds. If not, Sim aborts the protocol on behalf of the honest party. After completing the simulation, Sim outputs the adversary's view.

viii. If $\mathsf{MACCheck} = 1$, Sim aborts the simulation.

Figure 26: The simulator for the verification phase.

$\mathbf{Hyb}_0$: In this hybrid, Sim receives the honest parties' inputs and runs the protocol honestly. This corresponds to the real-world scenario.

$\mathbf{Hyb}_1$: In this hybrid, Sim additionally sets $\mathsf{CompCheck} = \mathsf{MACCheck} = 0$ at the beginning. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ have the same output distribution.

$\mathbf{Hyb}_2$: In this hybrid, Sim does not samples each honest party P_i's shares of each partially authenticated triple $[\![a]\!], [\![b]\!], [c]$ consumed for generating authenticated sharings for his inputs while emulating $\mathcal{F}_{\mathsf{prep}}$. Instead, Sim samples them when they are used for computation, i.e., when P_i needs to verify $[a], [b], [c]$ during the authentication process for his inputs. This does not affect the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

$\mathbf{Hyb}_3$: In this hybrid, while authenticating inputs, for each input x of an honest party P_i, Sim does not follow the protocol to verify the sharings $[a], [b], [c]$. Instead, he verifies whether the sum of corrupted parties' shares of $[a]$ (resp. $[b], [c]$) received from them is the same as the sum of corrupted parties' shares of $[a]$ (resp. $[b], [b]$) sent to them by Sim while emulating $\mathcal{F}_{\mathsf{prep}}$. This only changes the output distribution if

the adversary attaches errors e_a, e_b, e_c to the sum of the corrupted parties' shares of $[a], [b], [c]$ respectively, with e_a, e_b, e_c not all zero. Since a, b is sampled randomly after e_a, e_b, e_c are determined, the probability that $e_c = e_a \cdot b + e_b \cdot a + e_a \cdot e_b$ holds is no more than $1/|\mathbb{F}|$, which is negligible. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same output distribution.

$\mathbf{Hyb}_4$: In this hybrid, while authenticating inputs, Sim does not generate honest parties' shares of $[a]$ in the partially authenticated triples used for authenticating honest parties' inputs. Instead, for each input x of an honest party P_i, Sim samples a random element as $x + a$ and then samples honest parties' shares of $[a]$ based on $x, x + a$ and corrupted parties' shares of $[a]$. Since Sim samples a randomly and then samples honest parties' shares of $[a]$ based on a and corrupted parties' shares of $[a]$ while faithfully emulating $\mathcal{F}_{\mathsf{prep}}$, we only change the way a is sampled. Since a is uniformly random in $\mathbb{F}$, so is $x + a$. Thus, we only change the order of generating a and $x + a$ without changing their distributions. Thus, $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ have the same output distribution.

$\mathbf{Hyb}_5$: In this hybrid, whenever a corrupted party needs to send a message to all the parties, Sim regards that all parties receive the same message as the actual message received from P_h. These messages are only used for additional computations later, but are not used for computing the transcripts sent from honest parties to the corrupted parties. This does not affect the output distribution. Thus, $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same output distribution.

$\mathbf{Hyb}_6$: In this hybrid, for each multiplication gate of C' (with inputs $x_\alpha^{(j)}, y_\alpha^{(j)}$), Sim does not sample P_h's shares of $[\![a_\alpha]\!], [\![b_\alpha]\!]$ in the j-th length-k tensor product while emulating $\mathcal{F}_{\mathsf{prep}}$. Instead, Sim samples P_h's shares of $[x_\alpha^{(j)} + a_\alpha], [y_\alpha^{(j)} + b_\alpha]$ randomly during the evaluation phase and then computes his shares of $[a_\alpha] = [x_\alpha^{(j)} + a_\alpha] - [x_\alpha^{(j)}]$ and $[b_\alpha] = [y_\alpha^{(j)} + b_\alpha] - [y_\alpha^{(j)}]$. After that, a_α, b_α are computed accordingly and then be used to compute P_h's shares of $[\Delta \cdot a_\alpha], [\Delta \cdot b_\alpha]$. Since P_h's shares of $[a_\alpha], [b_\alpha]$ are uniformly random in $\mathbb{F}$, so are his shares of $[x_\alpha^{(j)} + a_\alpha], [y_\alpha^{(j)} + b_\alpha]$. Thus, we only change the order of generating P_h's shares of $[a_\alpha], [b_\alpha]$ and $[x_\alpha^{(j)} + a_\alpha], [y_\alpha^{(j)} + b_\alpha]$ without changing their distributions. Thus, $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ have the same output distribution.

$\mathbf{Hyb}_7$: In this hybrid, during the evaluation, for each multiplication gate of C' (with inputs $x_\alpha^{(j)}, y_\alpha^{(j)}$), Sim additionally follows the protocol to compute the shares of $[x_\alpha^{(j)} + a_\alpha], [y_\alpha^{(j)} + b_\alpha]$ for each party $P_i \neq P_h$ using the regarded values (in $\mathbf{Hyb}_5$) sent from corrupted parties. These shares, together with the sampled shares for P_h, are used for reconstructing $x_\alpha^{(j)} + a_\alpha, y_\alpha^{(j)} + b_\alpha$ additionally.

Then, after receiving $\overline{x_\alpha^{(j)} + a_\alpha}, \overline{y_\alpha^{(j)} + b_\alpha}$,

- If $x_\alpha^{(j)}$ (resp. $y_\alpha^{(j)}$) can be computed directly from input wires of C', Sim additionally sets $\mathsf{MACCheck} = 1$ if $\overline{x_\alpha^{(j)} + a_\alpha} \neq x_\alpha^{(j)} + a_\alpha$ (resp. $\overline{y_\alpha^{(j)} + b_\alpha} \neq y_\alpha^{(j)} + b_\alpha$).
- Otherwise, Sim additionally computes $\delta_{x_\alpha^{(j)}} = \overline{x_\alpha^{(j)} + a_\alpha} - (x_\alpha^{(j)} + a_\alpha)$ (resp. $\delta_{y_\alpha^{(j)}} = \overline{y_\alpha^{(j)} + b_\alpha} - (y_\alpha^{(j)} + b_\alpha)$). If $\overline{x_\alpha^{(j)} + a_\alpha} \neq x_\alpha^{(j)} + a_\alpha$ (resp. $\overline{y_\alpha^{(j)} + b_\alpha} \neq y_\alpha^{(j)} + b_\alpha$), Sim sets $\mathsf{CompCheck} = 1$.

This does not affect the output distribution. Thus, $\mathbf{Hyb}_7$ and $\mathbf{Hyb}_6$ have the same output distribution.

$\mathbf{Hyb}_8$: In this hybrid, for each output wire of C' (with wire value v), Sim does not sample P_h's share of the consumed sharing $[\![r]\!]$ for authenticating the output value while emulating $\mathcal{F}_{\mathsf{prep}}$. Instead, Sim samples P_h's share of $[r + v]$ randomly during the evaluation phase and then computes his share of $[r] = [r + v] - [v]$. After that, r are computed accordingly and then be used to compute P_h's share of $[\Delta \cdot r]$. For the same reason as in $\mathbf{Hyb}_6$, $\mathbf{Hyb}_8$ and $\mathbf{Hyb}_7$ have the same output distribution.

$\mathbf{Hyb}_9$: In this hybrid, during the evaluation, for each output wire of C' (with wire value v), Sim additionally follows the protocol to compute the shares of $[r + v]$ for each party $P_i \neq P_h$ using the regarded values (in $\mathbf{Hyb}_5$) sent from corrupted parties. These shares, together with the sampled shares for P_h, are used for reconstructing $r + v$ additionally.

Then, after receiving $\overline{r + v}$,

- If v an be computed directly from input wires of C', Sim additionally sets $\mathsf{MACCheck} = 1$ if $\overline{r + v} \neq r + v$.

- Otherwise, Sim additionally computes $\delta_v = \overline{r+v} - (r+v)$. If $\overline{r+v} \neq r+v$, Sim sets $\mathsf{CompCheck} = 1$.

This does not affect the output distribution. Thus, $\mathbf{Hyb}_9$ and $\mathbf{Hyb}_8$ have the same output distribution.

$\mathbf{Hyb}_{10}$: In this hybrid, during the evaluation, while simulating $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, if $\mathsf{MACCheck} = 0$, Sim does not follow the protocol to compute P_h's share of $[\sigma]$. Instead, Sim computes the P_h's share of $[\sigma]$ based on the other parties' shares (computed using regarded messages) and $\sum_{i=1}^{n} \sigma_i = 0$. Note that when $\mathsf{MACCheck} = 0$, all the opened values are opened correctly via Π_{Open}. In this case, it must hold that $[\sigma] = [0]$, so we only change the way of generating P_h's share of $[\sigma]$ without changing its distribution. Thus, $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ have the same output distribution.

$\mathbf{Hyb}_{11}$: In this hybrid, during the evaluation, while simulating $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, if $\mathsf{MACCheck} = 1$, Sim doesn't directly compute the honest party P_h's share of $[\Delta \cdot A_i]$ for each opened value A_i. Instead, Sim first computes each $\Delta \cdot A_i$, and then Sim computes the honest party's share of $[\Delta \cdot A_i]$ based on the corrupted parties' shares and $\Delta \cdot A_i$. We just change the way the honest party's share of each $[\Delta \cdot A_i]$ is generated without changing its distribution. Thus, $\mathbf{Hyb}_{11}$ and $\mathbf{Hyb}_{10}$ have the same output distribution.

$\mathbf{Hyb}_{12}$: In this hybrid, during the evaluation, while simulating $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, after following the protocol to check whether $\sum_{i=1}^{n} \sigma_i = 0$, Sim aborts the simulation if $\mathsf{MACCheck} = 1$. The only difference between $\mathbf{Hyb}_{12}$ and $\mathbf{Hyb}_{11}$ is that the corrupted parties may not open all the values of SPDZ sharing checked in this execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ correctly, but the verification of MACs may pass.

We suppose that $A_1, \ldots, A_m$ are opened with additive errors $\delta^{(1)}, \ldots, \delta^{(m)}$, and the sum of all the committed shares $[\sigma]$ of corrupted parties is with an additive error δ_σ. Then

$$\begin{aligned}\sigma =& \sum_{i\in[1,n],i\neq h} \left(\sum_{\alpha=1}^{m} \chi^\alpha \cdot (\Delta \cdot A_\alpha)^{(i)} - \Delta_i \cdot \sum_{\alpha=1}^{m} \chi^\alpha \cdot A_k \right) + \delta_\sigma \\ &+ \left(\sum_{\alpha=1}^{m} \chi^\alpha \cdot (\Delta \cdot A_\alpha)^{(h)} - \Delta_h \cdot \sum_{\alpha=1}^{m} \chi^\alpha \cdot (A_\alpha + \delta^{(\alpha)}) \right) \\ =& \delta_\sigma + \Delta_h \cdot \sum_{\alpha=1}^{m} \chi^\alpha \cdot \delta^{(\alpha)}.\end{aligned}$$

Here $(\Delta \cdot A_\alpha)^{(i)}$ is used to denote P_i's share of $[\Delta \cdot A_\alpha]$. Since Δ_h is not used in computing any message sent to the corrupted parties at this moment, $\Delta_h \cdot \sum_{\alpha=1}^{m} \chi^\alpha \cdot \delta^{(\alpha)}$ is a random element in $\mathbb{F}$ as long as $\sum_{\alpha=1}^{m} \chi^\alpha \cdot \delta^{(\alpha)}$ is non-zero. When $(\delta^{(1)}, \ldots, \delta^{(m)})$ is a nonzero vector, since a non-zero degree-m polynomial only has at most m zeros, so $\sum_{\alpha=1}^{m} \chi^\alpha \cdot \delta^{(\alpha)} = 0$ only happens with at most $m/|\mathbb{F}|$ probability. Since $m = \mathsf{poly}(n, \kappa)$, the probability is negligible. On the other hand, in the case where $\sum_{\alpha=1}^{m} \chi^\alpha \cdot \delta^{(\alpha)} = 0$, σ is opened to a random element in $\mathbb{F}$, which equals 0 with $1/|\mathbb{F}|$ probability, which is also negligible.

Thus, the distributions of $\mathbf{Hyb}_{12}$ and $\mathbf{Hyb}_{11}$ are statistically close.

$\mathbf{Hyb}_{13}$: In this hybrid, during the reduction to inner-product check, Sim additionally computes the error $\delta_z = \sum_{j=1}^{M} s^j \delta_{z_j}$ on z for the sharing $[\![z]\!]$, where δ_{z_j} is the additive error attached to the value z_j computed during the evaluation phase. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{13}$ and $\mathbf{Hyb}_{12}$ have the same output distribution.

$\mathbf{Hyb}_{14}$: In this hybrid, during the reduction to inner-product check, after computing δ_z, Sim aborts the simulation if $\mathsf{ErrorCheck} = 1$ but $\delta_z = 0$. This only changes the output distribution when there is some $j \in \{1, \ldots, M\}$ such that $\delta_{z_j} \neq 0$, but it happens that $\delta_z = 0$.

Since s is randomly determined after $\delta_{z_1}, \ldots, \delta_{z_M}$ are all fixed, this only happens when s is a solution for the equation $\sum_{i=1}^{M} s^i \cdot \delta_{z_i} = 0$. Since the left-hand side is a degree-M polynomial in s, it has at most $M = \mathsf{poly}(\kappa)$ zeros. Therefore, the probability that $\delta_z = 0$ is at most $M/|\mathbb{F}|$, which is negligible. Thus, the distributions of $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{13}$ are statistically close.

$\mathbf{Hyb}_{15}$: In this hybrid, during the separation into blocks, for the sharing $[\![r^{(j)}]\!]$ $(j = 1, \ldots, \ell - 1)$ used to authenticate $[z^{(j)}]$, Sim does not generate P_h's share of it while emulating $\mathcal{F}_{\mathsf{prep}}$. Instead, Sim samples P_h's share of $[r^{(j)} + z^{(j)}]$ randomly and then computes his share of $[r^{(j)}] = [r^{(j)} + z^{(j)}] - [z^{(j)}]$. Then, P_h's share of $[\Delta \cdot r^{(j)}]$ is computed based on $\Delta, r^{(j)}$ and other parties' shares. Note that P_h's share of $[r^{(j)}]$ is sampled

randomly in $\mathcal{F}_{\mathsf{prep}}$, so his share of $[r^{(j)} + z^{(j)}]$ is also uniformly random. Therefore, we only change the order of generating P_h's shares of $[r^{(j)} + z^{(j)}]$ and $[r^{(j)}]$ without changing their distributions. Thus, $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$ have the same output distribution.

$\mathbf{Hyb}_{16}$: In this hybrid, during the first reduction in vector length, after receiving $\overline{r^{(j)} + z^{(j)}}$ for each $j = 1, \ldots, \ell - 1$ on behalf of P_h, Sim regards that all parties receive without changing the transcripts. Then, Sim reconstructs $r^{(j)} + z^{(j)}$ from all parties' shares (obtained by using the regarded messages to run the protocol) and additionally computes $\delta_{z^{(j)}} = \overline{r^{(j)} + z^{(j)}} - (r^{(j)} + z^{(j)})$. Finally, Sim computes $\delta_{z^{(\ell)}} = \delta_z - \sum_{j=1}^{\ell-1} \delta_{z^{(j)}}$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{16}$ and $\mathbf{Hyb}_{15}$ have the same output distribution.

$\mathbf{Hyb}_{17}$: In this hybrid, before doing the recursive check for each group of multiplication gates (with index j), Sim additionally initializes $\delta_w = \delta_{z^{(j)}}$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{17}$ and $\mathbf{Hyb}_{16}$ have the same output distribution.

Let $\mathbf{Hyb}_{18.0.5} = \mathbf{Hyb}_{17}$. We then construct $\mathbf{Hyb}_{18.\gamma.1}, \ldots, \mathbf{Hyb}_{15.\gamma.5}$ ($\gamma = 1, \ldots, \lceil \log_\lambda k \rceil$) for the γ-th reduction in vector length.

$\mathbf{Hyb}_{18.\gamma.1}$: In this hybrid, during the γ-th reduction in vector length for each group of multiplication gates, for the sharing $[\![r_i]\!]$ ($i = 1, \ldots, \lambda - 1$) used to authenticate $[w_i]$, Sim does not generate P_h's share of it while emulating $\mathcal{F}_{\mathsf{prep}}$. Instead, Sim samples P_h's share of $[r_i + w_i]$ randomly and then computes his share of $[r_i] = [r_i + w_i] - [w_i]$. Then, P_h's share of $[\Delta \cdot r_i]$ is computed based on Δ, r_i and other parties' shares. Note that P_h's share of $[r_i]$ is sampled randomly in $\mathcal{F}_{\mathsf{prep}}$, so his share of $[r_i + w_i]$ is also uniformly random. Therefore, we only change the order of generating P_h's shares of $[r_i + w_i]$ and $[r_i]$ without changing their distributions. Thus, $\mathbf{Hyb}_{18.\gamma.1}$ and $\mathbf{Hyb}_{18.(\gamma-1).5}$ have the same output distribution.

$\mathbf{Hyb}_{18.\gamma.2}$: In this hybrid, during the γ-th reduction in vector length for each group of multiplication gates, after receiving $\overline{r_i + w_i}$ for each $i = 1, \ldots, \lambda - 1$ on behalf of P_h, Sim regards that all parties receive without changing the transcripts. Then, Sim reconstructs $r_i + w_i$ from all parties' shares (obtained by using the regarded messages to run the protocol) and additionally computes $\delta_{w_i} = \overline{r_i + w_i} - (r_i + w_i)$. Finally, Sim computes $\delta_{w_\lambda} = \delta_w - \sum_{i=1}^{\lambda-1} \delta_{w_i}$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{18.\gamma.2}$ and $\mathbf{Hyb}_{18.\gamma.1}$ have the same output distribution.

$\mathbf{Hyb}_{18.\gamma.3}$: In this hybrid, during the γ-th reduction in vector length for each group of multiplication gates, for the sharing $[\![r_i]\!]$ ($i = \lambda+1, \ldots, 2\lambda-1$) used to authenticate $[h(i)]$, Sim does not generate P_h's share of it while emulating $\mathcal{F}_{\mathsf{prep}}$. Instead, Sim samples P_h's share of $[r_i + h(i)]$ randomly and then computes his share of $[r_i] = [r_i + h(i)] - [h(i)]$. Then, P_h's share of $[\Delta \cdot r_i]$ is computed based on Δ, r_i and other parties' shares. Note that P_h's share of $[r_i]$ is sampled randomly in $\mathcal{F}_{\mathsf{prep}}$, so his share of $[r_i + h(i)]$ is also uniformly random. Therefore, we only change the order of generating P_h's shares of $[r_i+h(i)]$ and $[r_i]$ without changing their distributions. Thus, $\mathbf{Hyb}_{18.\gamma.3}$ and $\mathbf{Hyb}_{18.\gamma.2}$ have the same output distribution.

$\mathbf{Hyb}_{18.\gamma.4}$: In this hybrid, during the first reduction in vector length for each group of multiplication gates, after receiving $\overline{r_i + h(i)}$ for each $i = \lambda+1, \ldots, 2\lambda-1$ on behalf of P_h, Sim regards that all parties receive without changing the transcripts. Then, Sim reconstructs $r_i+h(i)$ from all parties' shares (obtained by using the regarded messages to run the protocol) and additionally computes $\delta_{h(i)} = \overline{r_i + h(i)} - (r_i + h(i))$. Finally, Sim computes $\delta_{h(r)} = L'(\delta_{w_1}, \ldots, \delta_{w_\lambda}, \delta_{h(\lambda+1)}, \ldots, \delta_{h(2\lambda-1)})$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{18.\gamma.4}$ and $\mathbf{Hyb}_{18.\gamma.3}$ have the same output distribution.

$\mathbf{Hyb}_{18.\gamma.5}$: In this hybrid, during the first reduction in vector length for each group of multiplication gates, if $\delta_w \neq 0$ but $\delta_{h(x)} = 0$, Sim aborts the simulation. Otherwise, Sim updates $\delta_w = \delta_{h(x)}$. This only changes the output distribution if δ_w is non-zero but $\delta_{h(x)} = 0$.

Note that $\sum_{i=1}^{\lambda} \delta_{w_i} = \delta_w$, so $\delta_w \neq 0$ implies that $(\delta_{w_1}, \ldots, \delta_{w_\lambda})$ is a non-zero vector. Thus, the degree-$(2\lambda - 2)$ polynomial $h_\delta(\cdot)$ with $h_\delta(i) = z_i$ for $i = 1, \ldots, \lambda$ and $h_\delta(i) = h(i)$ for $i = \lambda + 1, \ldots, 2\lambda - 1$ is a non-zero polynomial. Thus, $h_\delta(\cdot)$ only has at most $2k - 2$ zeros. Since $\delta_{h(r)} = h_\delta(r)$ for a random r, the probability that $\delta_{h(r)}$ is at most $(2\lambda - 2)/|\mathbb{F}|$, which is negligible. Thus, the distributions of $\mathbf{Hyb}_{15.\gamma.5}$ and $\mathbf{Hyb}_{15.\gamma.4}$ are statistically close.

$\mathbf{Hyb}_{19}$: In this hybrid, after obtaining the final triple $([\![u]\!], [\![v]\!], [\![w]\!])$ for each group of multiplication gates during the verification phase, Sim does not sample P_h's shares of $[\![a]\!], [\![b]\!]$ for the consumed triple while emulating $\mathcal{F}_{\mathsf{prep}}$ and uses them to compute his shares of $[\![u+a]\!], [\![v+b]\!]$ here. Instead, Sim samples P_h's shares of $[u + a], [v + b]$ randomly during the evaluation phase and then computes his shares of $[a] = [u + a] - [u]$

and $[b] = [v + b] - [v]$. After that, a, b are computed accordingly and then be used to compute P_h's shares of $[\Delta \cdot a], [\Delta \cdot b], [\![c]\!]$. Since P_h's shares of $[a], [b]$ are uniformly random in $\mathbb{F}$, so are his shares of $[u + a], [v + b]$. Thus, we only change the order of generating P_h's shares of $[a], [b]$ and $[u + a], [v + b]$ without changing their distributions. Thus, $\mathbf{Hyb}_{19}$ and $\mathbf{Hyb}_{18.\lceil \log_\lambda k \rceil .5}$ have the same output distribution.

$\mathbf{Hyb}_{20}$: In this hybrid, after obtaining $\overline{u + a}, \overline{v + b}$ for each group of multiplication gates during the verification phase, Sim additionally follows the protocol to compute the shares of $[u + a], [v + b]$ for each party $P_i \neq P_h$ using the regarded values sent from corrupted parties. These shares, together with the sampled shares for P_h, are used for reconstructing $u + a, v + b$ additionally. Then, after receiving $\overline{u + a}, \overline{v + b}$, Sim additionally sets $\mathsf{MACCheck} = 1$ if $\overline{u + a} \neq u + a$ or $\overline{v + b} \neq v + b$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{20}$ and $\mathbf{Hyb}_{19}$ have the same output distribution.

$\mathbf{Hyb}_{21}$: In this hybrid, after obtaining $\overline{u + a}, \overline{v + b}$ for each group of multiplication gates during the verification phase, if $\mathsf{MACCheck} = 1$ or $\delta_w \neq 0$, Sim does not follow the protocol to compute $[w]$. Instead, Sim computes $w = uv + \delta_w$ and then computes P_h's share of $[w]$ based on corrupted parties' shares and the secret. Since δ_w is the pre-computed additive error attached to w, the sharing $[w]$ should be a sharing of the secret $uv + \delta_w$. We only change the way of generating P_h's share of $[w]$ without changing its distribution. Thus, $\mathbf{Hyb}_{21}$ and $\mathbf{Hyb}_{20}$ have the same output distribution.

$\mathbf{Hyb}_{22}$: In this hybrid, after obtaining $\overline{u + a}, \overline{v + b}$ for each group of multiplication gates during the verification phase, if $\mathsf{MACCheck} = \delta_w = 0$, Sim does not follow the protocol to compute P_h's share of $[\tau]$. Instead, Sim additionally follows the protocol to compute the share of $[\tau]$ for each party $P_i \neq P_h$ using the regarded values sent from corrupted parties. Then, P_h's share is computed based on the secret $\tau = 0$ and other parties' shares. Note that in this case, $([\![u]\!], [\![v]\!], [\![w]\!])$ is a valid triple, and $[\tau] = [uv - w]$ is computed honestly using the other triple $([\![a]\!], [\![b]\!], [\![c]\!])$, so it must be a sharing of secret 0. Therefore, we only change the way Sim generates P_h's share of $[\tau]$ without changing its distribution. Thus, $\mathbf{Hyb}_{22}$ and $\mathbf{Hyb}_{21}$ have the same output distribution.

$\mathbf{Hyb}_{23}$: In this hybrid, after obtaining $\overline{\tau}$ from P_{king} during the verification phase, if $\tau \neq \overline{\tau}$ (where τ is reconstructed from all parties' shares), Sim additionally sets $\mathsf{MACCheck} = 1$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{22}$ and $\mathbf{Hyb}_{23}$ have the same output distribution.

$\mathbf{Hyb}_{24}$: In this hybrid, at the end of the verification, Sim simulates the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as the execution in the evaluation phase. For the same reason as in $\mathbf{Hyb}_{10}, \mathbf{Hyb}_{11}, \mathbf{Hyb}_{12}$, the distributions of $\mathbf{Hyb}_{24}$ and $\mathbf{Hyb}_{23}$ are statistically close.

$\mathbf{Hyb}_{25}$: In this hybrid, during the output reconstruction, Sim does not follow the protocol to compute $[\![v]\!]$ for each party's output wire value v of C'. Instead, Sim evaluates the circuit C with the parties' inputs to get v and then computes P_h's share of $[v]$ based on the secret and other parties' shares. Then, v is reconstructed from all parties' shares, and P_h's share of $[\Delta \cdot v]$ is computed from Δ, v together with other parties' shares.

Note that when the protocol is not aborted before the output reconstruction, then it is guaranteed that all authenticated wire values in C' are computed honestly from the multiplication outputs from previous layers. If not, $\mathsf{CompCheck} = 1$. This implies that $\delta_z \neq 0$. Since $\delta_z = \sum_{j=1}^{\ell} \delta_{z^{(j)}}$, there exists one j such that $\delta_{z^{(j)}}$, implying the $\delta_w \neq 0$ for the final triple $([\![u]\!], [\![v]\!], [\![w]\!])$ for checking the j-th group of multiplication gates. In this case, the protocol or simulation must be aborted, which leads to a contradiction.

Therefore, each output v of C' is honestly computed from the circuit inputs, so we only change the way of generating P_h's share of $[v]$ without changing its distribution. Similarly, P_h's share of $[\Delta \cdot v]$ is determined by the secret and other parties' shares, i.e. determined by P_h's share of $[v]$, which means that the distribution of P_j's share of $[\![v]\!]$ remains unchanged. Thus, $\mathbf{Hyb}_{25}$ and $\mathbf{Hyb}_{24}$ have the same output distribution.

$\mathbf{Hyb}_{26}$: In this hybrid, during the output reconstruction, for each output wire value v of C', Sim additionally checks whether the reconstructed value $\overline{v}$ equals v. If not, Sim additionally sets $\mathsf{MACCheck} = 1$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{26}$ and $\mathbf{Hyb}_{25}$ have the same output distribution.

$\mathbf{Hyb}_{27}$: In this hybrid, during the output reconstruction, Sim simulates the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as the execution in the evaluation phase. For the same reason as in $\mathbf{Hyb}_{10}, \mathbf{Hyb}_{11}, \mathbf{Hyb}_{12}$, the distributions of $\mathbf{Hyb}_{27}$ and $\mathbf{Hyb}_{26}$ are statistically close.

$\mathbf{Hyb}_{28}$: In this hybrid, when emulating $\mathcal{F}_{\mathsf{prep}}$, Sim delays the generation of the P_h's shares when they are needed. This does not change the output distribution. Thus, $\mathbf{Hyb}_{28}$ and $\mathbf{Hyb}_{27}$ have the same output

distribution.

Note that if the protocol does not abort at the end of the execution and Sim does not abort the simulation, the honest party's shares are not used during the interaction with corrupted parties. Furthermore, honest parties' inputs to C' are only used for computing the outputs of C' in this case.

$\mathbf{Hyb}_{29}$: In this hybrid, we change the preparation of the honest party's output when the protocol does not abort at the end of the execution and Sim does not abort the simulation. Sim no longer generates the P_h's output shares when emulating $\mathcal{F}_{\mathsf{prep}}$ and no longer generates honest parties' inputs. Instead, Sim obtains corrupted parties' outputs of C from $\mathcal{F}$ and then uses their inputs to C' (which includes the masks for outputs of C) to compute their outputs of C'. For honest parties' outputs of C', Sim directly samples random values as these outputs. Note that the masks are randomly sampled, so the honest parties' outputs of C' are also random. By the correctness of the construction, $\mathbf{Hyb}_{29}$ and $\mathbf{Hyb}_{28}$ have same output distribution.

$\mathbf{Hyb}_{30}$: In this hybrid, Sim does not sample honest parties' outputs of C' and then use them to compute P_h's shares of them. Instead, Sim directly samples P_h's shares. Since the outputs are random, so are P_h's shares. Therefore, we only change the order of generating v and P_h's share of $[v]$ for each output v of C' for honest parties without changing their distributions. Furthermore, Sim only generates P_h's intermediate shares when they are needed. If the protocol is not aborted, then they are not required to be computed. This also does not affect the output distribution. Thus, $\mathbf{Hyb}_{30}$ and $\mathbf{Hyb}_{30}$ have the same output distribution.

Note that $\mathbf{Hyb}_{30}$ is the ideal-world scenario. Thus, Π_{main} computes $\mathcal{F}$ with statistical security.

I Cost Analysis

We analyze the communication cost of our main protocol as follows. The cost is measured by field elements, and we assume that $m_I, m_O = O(|C|)$. We first analyze the communication cost in the $\{\mathcal{F}^k_{\mathsf{patensor}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model ($\mathcal{F}_{\mathsf{prep}}$ is instantiated with Π_{prep}), without instantiating the functionalities.

- **Preprocessing Phase.**
  - **Random Values:** The preprocessing of random values only involves invocations of $\mathcal{F}^k_{\mathsf{patensor}}$ and local computation, which requires no additional communication.
  - **Input Masks:** Similar to the preprocessing of random authenticated sharings, this step requires no communication.
  - **Authenticated Triples:** Authenticating $[c], [c']$ during the preparation for each authenticated triple requires $4n$ communication. During the verification of each triple, two values d, e need to be opened. Thus, the amortized communication cost is $8n$ per authenticated triple. Since $m_A = \ell = O(|C|/k)$, $O(|C|n/k)$ total communication cost is required.
  - **Tensor Products:** Similar to the preprocessing of random authenticated sharings, this step requires no communication.
- **Evaluation Phase.**
  1. **Authenticating Inputs:** The servers need to send 3 additive sharings to a client for each input wire of C' attached to this client, and the client will send an element back to all the servers. This requires $4n$ communication in total. Since the number of input wires in C' is $m_I + m_O$, the communication cost of this step is $4(m_I + m_O)n$.
  2. **Evaluating the Circuit:** The communication cost during the evaluation of the circuit is the same as that of the semi-honest SPDZ, which is $4|C|n$. Then, authenticating the output values of C' costs communication of another $2m_O n$ field elements.
  3. **MAC Check:** This step only involves invocations of $\mathcal{F}_{\mathsf{Coin}}$ and $\mathcal{F}_{\mathsf{Commit}}$.
- **Verification Phase.**

1. **Reducing to Inner-Product Check:** This reduction only involves an invocation of $\mathcal{F}_{\mathsf{Coin}}$ and local computations, which do not require any additional communication.
2. **Separate into Blocks:** This step requires authentication of the sharings $[z^{(1)}], \ldots, [z^{(\ell-1)}]$, which requires $2(\ell-1)n = O(|C|n/k)$ communication cost.
3. **Recursive Check:** During each reduction in vector length, the sharings $[w_1], \ldots, [w_{\lambda-1}]$ and $[h(\lambda+1), \ldots, h(2\lambda-1)]$ needs to be authenticated, which requires $2(\lambda-1)$ authentication steps. Since the reduction needs to be performed $\ell \cdot \log_\lambda k$ times, there are in total $2\ell(\lambda-1) \cdot \log_\lambda k$ authentication steps, which requires communication of $4n\ell(\lambda-1) \cdot \log_\lambda k = O(|C|n\lambda k^{-1} \cdot \log_\lambda k)$ field elements.
4. **Verify the Final Triple.** During the check of each final triple, three values need to be opened via Π_{Open}, which requires $6n$ communication cost. Thus, the total communication cost of this step is $6\ell n = O(|C|n/k)$.

- **Output Reconstruction.** For each output wire of C', the parties need to publicly reconstruct the wire value, which requires $2n$ communication. Since the number of output wires in C' is the same as the number of output wires in C. The communication cost of this step is $2m_O$.

The instantiations of $\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}$ require $n^2(\mathsf{poly}(\kappa))$ communication. Therefore, regardless of the communication cost from PCGs, the total communication cost is $4|C|n + 4m_I n + 8m_O n + O(|C|n\lambda k^{-1} \cdot \log_\lambda k) + n^2\mathsf{poly}(\kappa))$ field elements.

Regardless of the cost of PCGs, the computational cost during the preprocessing phase mainly comes from the verification on the length-k tensor product correlations (i.e., Steps 6 and 7 in $\Pi^k_{\mathsf{patensor}}$), which requires $O(|C|nk)$ field operations. During the evaluation phase and the output reconstruction, the computational cost is clearly $O(|C|n)$. During the verification, the computational cost mainly comes from the computation of $[\boldsymbol{x}^{(j)} \otimes \boldsymbol{y}^{(j)}]$ and the recursive check. The former requires $O(|C|nk)$ field operations, and the latter requires $O(n\ell'^2\lambda^2)$ for each reduction in vector length, resulting in a total computational cost of $O(|C|nk)$ field operations. Thus, the total computational cost (beyond PCG) is $O(|C|nk)$.

If considering the total cost including the PCGs, there is a trade-off between the communication overhead (the $o(|C|)$ additive term) and the total computational complexity. If we want low communication overhead, we may choose a large k and instantiate $\mathcal{F}^k_{\mathsf{patensor}}$ using PCGs for 2-party tensor product correlations. For example, when we choose $k = |C|/\log|C|$, then the total communication overhead becomes $n^2 \cdot \mathsf{poly}(\log|C|) + n^2\mathsf{poly}(\kappa)$, which matches the semi-honest baseline. Note that the PCG from [BCG+20] can instantiate $\mathcal{F}^{\mathsf{prog}}_{\mathsf{tensor}}(m,k)$ from LPN-type assumptions with a seed size of $O(m \cdot \mathsf{poly}(\log k)/k)$ field elements and $O(mk \cdot \mathsf{poly}(\kappa, \log k))$ computational cost. $O(n^2)$ invocations of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{tensor}}(m_T + \lceil m_T/k \rceil + 1, k+1)$ and $O(m_T n)$ additional communication are required in total for instantiating the functionality $\mathcal{F}^k_{\mathsf{patensor}}$. Taking $m_T = O(|C|/k)$ as required in our main protocol, the additional communication cost does not affect the total communication cost, and the required PCG seeds have a total size of $O(|C| \cdot n^2 \cdot \mathsf{poly}(\log k)/k) = n^2 \cdot \mathsf{poly}(\log|C|)$. However, the total computational cost grows to $\tilde{O}(|C|^2 n^2)$.

On the other hand, if we require a low overall computational overhead, we may choose a smaller k, such as $k = \log|C|$, and apply the PCGs for n-party unauthenticated triples to instantiate $\mathcal{F}^k_{\mathsf{patensor}}$. Using the known results on PCGs for unauthenticated triples [BCG+20, ARR25], each invocation of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{uatrip}}(\gamma)$ requires a seed size of $O(n^2 \cdot \mathsf{poly}(\log\gamma))$ field elements and $O(n^2\gamma \cdot \mathsf{poly}(\kappa, \log\gamma))$ computational cost. $O(k^2) = O(\log^2|C|)$ invocations of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{uatrip}}(\gamma)$ with $\gamma = O(|C|/k) = O(|C|/\log|C|)$ are required to instantiate $\Pi^k_{\mathsf{uatensor}}$. Therefore, this approach only requires an $O(n^2|C|/\log|C|) + n^2\mathsf{poly}(\kappa)$ PCG seed size, and the total computational cost of $\tilde{O}(n^2|C|)$ matches the semi-honest baseline. However, the communication overhead becomes $O(|C|n^2/\log|C|) + n^2\mathsf{poly}(\kappa)$.

J The Functionality of the Preprocessing for Π_{turbo}

The functionality $\mathcal{F}_{\mathsf{Turboprep}}$ is presented in Figure 27 below.

Functionality $\mathcal{F}_{\mathsf{Turboprep}}$

On receiving $(\mathsf{Prep}, m_R, m_P, m_A, m_C)$ from all the honest parties, the trusted party run **Init**, **Random Values**, and **Authenticated Triples** of $\mathcal{F}_{\mathsf{prep}}$ and additional does the following:

1. Run steps 1 and 2 from **Random Values** of $\mathcal{F}_{\mathsf{prep}}$ $m = 2m_P + m_C$ times to create sharings $\{[\![w_i]\!]\}_{i=1}^{m}$.
2. For each pair of $(i,j) \in \{1,\ldots,m\}^2$, the trusted party computes $w_{i,j} = w_i \cdot w_j$ and receives corrupted parties' shares of $[w_{i,j}]$ from Sim. Then, the trusted party samples the honest parties' shares of it based on the secret and the corrupted parties' shares.
3. The trusted party distributes $\{[\![w_i]\!]\}_{i=1}^{m}, \{[w_{i,j}]\}_{i,j=1}^{m}$ to the parties.

Abort: If (abort, P) is received from Sim for an honest party P, the trusted party sends Δ to Sim and sends abort to P.

Figure 27: Functionality for the preprocessing phase of Π_{turbo}.

K The Verification Phase of Π_{turbo}

We present the verification phase Π_{turbover} of Π_{turbo} as follows in Figure 28.

Protocol Π_{turbover}

1. **Reducing to Inner-Product Check.** Denote all pairs of input wire values for multiplication gates in $|C|$ by $(x'_1, y'_1), \ldots, (x'_{|C|}, y'_{|C|})$ and all the output wire values of multiplication gates by $z'_1, \ldots, z'_{|C|}$, where each $x'_i = x_{i+m_I+m_O}$ and similar for y_i, z_i. Each z'_i is computed by $z'_i = x'_i \cdot y'_i$. Then:
   (a) The parties send $\mathsf{RandCoin}$ to $\mathcal{F}_{\mathsf{Coin}}$ and obtain a random element $s \in \mathbb{F}^*$.
   (b) The parties locally compute
   $$[\![\boldsymbol{x}]\!] = \left(\sum_{i=1}^{|C|} s^i [\![x'_1]\!], \ldots, \sum_{i=1}^{|C|} s^i [\![x'_{|C|}]\!]\right), \quad [\![\boldsymbol{y}]\!] = ([\![y'_1]\!], \ldots, [\![y'_{|C|}]\!])$$
   and
   $$[\![z]\!] = \sum_{i=1}^{|C|} s^i [\![z'_i]\!].$$
   With high probability, the coefficient $\sum_{i=1}^{|C|} s^i$ corresponding to each $[\![x'_j]\!] = [\![x_{j+m_I+m_O}]\!]$ in $[\![\boldsymbol{x}]\!]$ is non-zero for each $j = 1, \ldots, |C|$. If not, the parties abort the protocol.
2. **Perform the Inner-Product Check.** Assume that $x_i = \sum_{j=1}^{i-1} \beta_{j,i} \cdot z_j, y_i = \sum_{j=1}^{i-1} \gamma_{j,i} \cdot z_j$. The parties set $[\![a_i]\!] = \sum_{j=1}^{i_\alpha - 1} \beta_{j,i} \cdot [\![w_j]\!], [\![b_\alpha]\!] = \sum_{j=1}^{i_\alpha - 1} \gamma_{j,i} \cdot [\![w_j]\!]$. Then, the parties locally compute $[a_\alpha \cdot b_\beta] = \sum_{j_1=1}^{i-1} \sum_{j_2=1}^{i-1} \beta_{j,i}\gamma_{j,\beta}[w_{i,i}]$. The $|C|$ multiplication gates are regarded as evaluated using the length-$|C|$ partially authenticated tensor product $\{[\![a_\alpha]\!], [\![b_\alpha]\!]\}_{\alpha = m_I+m_O+1,\ldots,m_I+m_O+|C|}$, $\{[a_\alpha \cdot b_\beta]\}_{\alpha,\beta \in \{m_I+m_O+1,\ldots,m_I+m_O+|C|\}}$. Then, the parties run the second and the third steps of protocol Π_{IPcheck} (Figure 9) to verify that $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = z$, where $k = |C|$ and $\ell = 1$.

Figure 28: Protocol for the verification phase of Π_{turbo}.

L Protocol Description of Π_{turbo}

The protocol Π_{turbo} is presented in Figure 29 below.

Protocol Π_{turbo}

1. **Preprocessing.** Let m_I, m_O be the number of input and output wires in C respectively. The parties invoke $\mathcal{F}_{\mathsf{Turboprep}}$ with inputs $(\mathsf{Prep}, m_R, m_I + m_O, 1, m_C)$ to $\mathcal{F}_{\mathsf{Turboprep}}$ and receive the preprocessing data from $\mathcal{F}_{\mathsf{prep}}$, where $m_R = 2(\lambda - 1) \cdot \lceil \log_\lambda |C| \rceil$ and $m_C = |C|$.
2. **Evaluation.** The parties run $\Pi_{\mathsf{turboeval}}$.
3. **Verification.** The parties run Π_{turbover}.
4. **Output Reconstruction.** For each output wire with value v of C', the parties run Π_{Open} on $[\![v]\!]$. Then, the parties run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on all the opened output values and send each output wire value to the designated output party. Each party then checks whether the received outputs are all the same. If not, the party aborts the protocol. Otherwise, he locally computes his output of C from his output of C' and the masks he inputs to C'.

Figure 29: The protocol achieving $2n$ amortized communication cost per multiplication gate.

M Proof Sketch for the Security of Π_{turbo}

The main simulation strategy in proving the security of Π_{turbo} is similar to that of Π_{main}, so we focus on the difference between the two protocols, i.e.:

- The additional preprocessing of $[\![\boldsymbol{w}]\!], [\boldsymbol{w} \otimes \boldsymbol{w}]$.
- The online evaluation phase.

For the additional preprocessing, the only difference from the preprocessing of tensor products in Π_{tens} is that $\boldsymbol{a} = \boldsymbol{b} = \boldsymbol{w}$ during the additional preprocessing. Note that during the security proof of Π_{prep}, we do not really rely on the pseudorandomness of $\boldsymbol{b}$ to simulate the verification. Specifically, the honest party P_h's shares of $[x + x'], [y + y']$ can be sampled randomly by the simulator, which only relies on the fact that x', y' are independent pseudorandom elements. The commit-and-open process of $[\tau]$ and MAC check can be simulated in the same way as simulating Π_{tens}. Since the additional preprocessing process does not require any other communication, the protocol is computationally private.

For the correctness of $[\boldsymbol{w} \otimes \boldsymbol{w}]$, the attached error term to each $[w_i \cdot w_j]$ is linear to P_h's shares $w_i^{(h)}, w_j^{(h)}$ of $[w_i], [w_j]$, where the linear coefficients are determined before the pseudorandom vectors $\boldsymbol{r}_a = (r_a^{(1)}, \ldots, r_a^{(k)}), \boldsymbol{r}_b = (r_b^{(1)}, \ldots, r_b^{(k)})$ are obtained. Therefore, the total error term on τ contains $\langle \boldsymbol{r}_a \otimes \boldsymbol{r}_b, (\boldsymbol{w}^{(h)} \otimes \boldsymbol{\delta}_w + \boldsymbol{\delta}'_w \otimes \boldsymbol{w}^{(h)}) \rangle$ as we have analyzed in the proof of Π_{prep}, where $\boldsymbol{w}^{(h)}$ is P_h's share of $[\boldsymbol{w}]$, and $\boldsymbol{\delta}_w, \boldsymbol{\delta}'_w$ are vectors chosen by the adversary. From the pseudorandomness of $\boldsymbol{w}^{(h)}$, this error term is also pseudorandom if either of $\boldsymbol{\delta}_w, \boldsymbol{\delta}'_w$ is a non-zero vector. This guarantees that the existence of linear errors can be detected with an overwhelming probability.

For the security of online evaluation, only $z_i + w_i$ for $i = 1, \ldots, m$ are opened to the parties. Since each w_i is sampled randomly and independently by $\mathcal{F}_{\mathsf{prep}}$, all the communicated values during the evaluation phase are random. The simulator only needs to sample the honest party P_h's share of each $[z_i + w_i]$ randomly an sends it to P_{king}. Since we do not modify the verification framework, it can be simulated as in the simulation for Π_{main} with additional local computations provided in Π_{turbover}. The correctness of the evaluation can be guaranteed by the verification. Due to the above reasons, the statistical security also holds.