Dishonest Majority Constant-Round MPC with Linear Communication from DDH

Vipul Goyal
NTT Research
and Carnegie Mellon University
vipul@cmu.edu

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

Ankit Kumar Misra
University of California, Los Angeles
ankitkmisra@g.ucla.edu

Rafail Ostrovsky
University of California, Los Angeles
rafail@cs.ucla.edu

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

Chenkai Weng
Arizona State University
Chenkai.Weng@asu.edu

Abstract

In this work, we study constant round multiparty computation (MPC) for Boolean circuits against a fully malicious adversary who may control up to $n-1$ out of n parties. Without relying on fully homomorphic encryption (FHE), the best-known results in this setting are achieved by Wang et al. (CCS 2017) and Hazay et al. (ASIACRYPT 2017) based on garbled circuits, which require a quadratic communication in the number of parties $O(|C|\cdot n^2)$. In contrast, for non-constant round MPC, the recent result by Rachuri and Scholl (CRYPTO 2022) has achieved linear communication $O(|C|\cdot n)$.

In this work, we present the first concretely efficient constant round MPC protocol in this setting with linear communication in the number of parties $O(|C|\cdot n)$. Our construction can be based on any public-key encryption scheme that is linearly homomorphic for public keys. Our work gives a concrete instantiation from a variant of the El-Gamal Encryption Scheme assuming the DDH assumption. The analysis shows that when the computational security parameter $\lambda = 128$ and statistical security parameter $\kappa = 80$, our protocol achieves a smaller communication than Wang et al. (CCS 2017) when there are 16 parties for AES circuit and 8 parties for general Boolean circuits (where we assume that the numbers of AND gates and XOR gates are the same). When comparing with the recent work by Beck et al. (CCS 2023) that achieves constant communication complexity $O(|C|)$ in the strong honest majority setting ($t < (1/2-\epsilon)n$ where ϵ is a constant), our protocol is better as long as $n < 3500$ (when $t = n/4$ for their work).

1 Introduction

Secure multiparty computation (MPC) [Yao82, GMW87, CCD88, BGW88, RB89] allows a set of n parties to jointly compute a public function on their private inputs. The efficiency of MPC protocols can be measured from various aspects, and the most two common criteria are the communication complexity and the round complexity.

Communication-Efficient but Non-Constant Round MPC. The well-known SPDZ protocol was first introduced by Damgård et al. [DPSZ12], which achieves a very efficient online protocol whose communication complexity grows linearly with the number of parties in the dishonest majority setting. Due to its potential of

being used in practice, a long line of works [DKL+13, LOS14, KOS16, KPR18, BCS19], [BNO19, EGP+23] focus on improving both the offline phase and the online phase of the SPDZ protocol. Thanks to the recent progress of pseudo-random correlation generators (PCG) [BCG+19, BCG+20, WYKW21], Rachuri and Scholl [RS22] have achieved a linear communication complexity $O(|C|n)$ in both the offline phase and the online phase.

However, all SPDZ-style protocols suffer a large round complexity that grows linearly with the depth of the circuit. For circuits with large depths in the WAN setting, the network latency may become the main bottleneck.

Constant Round but Communication-Heavy MPC. Due to the round complexity of SPDZ protocols, another line of works target constant round MPC. Without relying on fully homomorphic encryption (which is not considered to be efficient in practice), the most common way is to follow the BMR template [BMR90] that generalizes the Yao's garbled circuits [Yao86] from the two-party setting to the multiparty setting. Despite the significant progress in [LPSY15, BLO16, BLO17], [HSS17, WRK17, HOSS18a, HOSS18b, YWZ20, BCO+21], the best-known result in the dishonest majority setting still requires a quadratic communication complexity in the number of parties $O(|C|n^2)$, which is insufficient to support applications that involve hundreds or thousands of parties.

The efficiency gap between these two types of MPC protocols leads to our following question:

"Can we construct a fully malicious MPC protocol in the dishonest majority setting that achieves the best in both worlds, i.e., with constant rounds and linear communication complexity?"

1.1 Our Contributions

In this work, we answer the above question affirmatively by presenting the first concretely efficient constant-round and fully malicious MPC protocol with linear communication in the dishonest majority setting ($t < n$).

Theorem 1. *Assuming DDH, LPN, and random oracles, there is a computationally secure constant-round MPC protocol against a fully malicious adversary controlling up to $n-1$ parties with communication of $O(|C|n\lambda)$ bits, where λ is the computational security parameter.*

Our construction has the following features.

- **Communication Complexity.** To compute a circuit C of size $|C|$, the total communication complexity of our protocol is $O(|C|n\lambda)$ bits, where λ is the computational security parameter.
- **Assumptions.** Our protocol makes a black-box use of building blocks in Le Mans [RS22], which can be instantiated based on the LPN assumption. Beyond the building blocks in Le Mans [RS22], the garbling phase of our construction only requires the DDH assumption and symmetric-key cryptographic assumptions (random oracle or pseudo-random generator).
- **Concrete Efficiency.** Comparing with the previously best-known results [WRK17, YWZ20] with quadratic communication, our protocol achieves a smaller communication as long as there are 16 parties for the AES-128 circuit when the computational security parameter $\lambda = 128$ and statistical security parameter $\kappa = 80$. We note that the work [BCO+21] can potentially achieve a linear communication when its underlying SPDZ preprocessing is instantiated by Le Mans [RS22]. Even after optimization, the communication cost of our protocol outperforms theirs by 11.7× on the AES-128 circuit and 11.3× on the SHA-256 circuit.

Our construction is conceptually simple. We note that the main difficulty in all previous works following the BMR template is to emulate the encryption algorithm of some symmetric-key encryption scheme where both the key k and the message m are secretly shared.

Our first idea is to replace the symmetric-key encryption scheme used in the BMR template by a public-key encryption scheme. In this way, while we still need to keep the private key secret, the public key can be

learnt by all parties. Now when emulating the encryption algorithm, only the message to be encrypted is secretly shared.

To allow all parties efficiently generate public-private key pairs (pk, k) where pk is known to all parties while k is (additively) shared among all parties, we make use of a public-key encryption scheme that is linearly homomorphic for public keys: For two key pairs $(\mathsf{pk}_1, k_1), (\mathsf{pk}_2, k_2)$, we require that there is a homomorphic operation $\tilde{+}$ such that $(\mathsf{pk}_1 \tilde{+} \mathsf{pk}_2, k_1 + k_2)$ is also a valid key pair. We show how key pairs can be efficiently generated relying on this property.

Although pk is known to all parties, emulating the encryption algorithm to encrypt a shared message m may still be inefficient. Our second idea is to let each party P_i just encrypt his message share. To be more concrete, suppose m is additively shared to all parties and m_i is held by P_i. We simply let each P_i encrypt m_i by pk and denote the cipher-text by ct_i. Note that a party having the private key k and all cipher-texts $(\mathsf{ct}_1, \ldots, \mathsf{ct}_n)$ can decrypt each m_i and compute m. This allows us to make use of the public-key encryption scheme in a black-box way. We show that this is sufficient for us to achieve a linear communication complexity.

2 Technical Overview and Related Works

We give a high-level overview of the main techniques used in this paper. Our goal is to construct a *constant-round* MPC protocol for a general Boolean circuit C consisting of AND and XOR gates. We focus on the dishonest majority setting, where up to $t = n - 1$ parties can be corrupted.

2.1 Background: Yao's Garbled Circuit and BMR Template

Most of the constant-round MPC protocols in the dishonest majority setting are based on multiparty garbling techniques derived from the well-known BMR protocol given by Beaver, Micali, and Rogaway [BMR90]. At a high level, the BMR technique is to let all parties jointly compute a Yao's garbled circuit of C. We first give a brief review of Yao's garbled circuits.

Review of Yao's Garbled Circuits. Yao's garbled circuit [Yao82] was designed in the two-party setting, where one party acts as the garbler to construct a garbled circuit, and the other party acts as the evaluator to evaluate this garbled circuit such that the evaluator only learns the circuit output and nothing else.

To garble a Boolean circuit C, the garbler first prepares a random bit value λ_w and a pair of labels $(k_{w,0}, k_{w,1})$ for each wire w in the circuit. During the evaluation phase, we want to maintain the invariant that the evaluator learns only $v_w \oplus \lambda_w$ and the corresponding label $k_{w,v_w \oplus \lambda_w}$, where v_w is the actual wire value, protected by the random bit λ_w. To this end, for a gate in C with input wires a, b and output wire c, we want the evaluator to be able to learn $(v_c \oplus \lambda_c, k_{c,v_c \oplus \lambda_c})$ if he holds $(v_a \oplus \lambda_a, k_{a,v_a \oplus \lambda_a})$ and $(v_b \oplus \lambda_b, k_{b,v_b \oplus \lambda_b})$. This can be done by preparing the following 4 ciphertexts. At a high level, we simply use the two labels, one from each input wire, as secret keys to encrypt the proper label for the output wire:

1. Let $f : \{0,1\}^2 \to \{0,1\}$ be the function computed by this gate, which is either the AND function or the XOR function. Let $g : \{0,1\}^2 \to \{0,1\}$ be defined by $g(x, y) = f(x \oplus \lambda_a, y \oplus \lambda_b) \oplus \lambda_c$. Then we have $g(v_a \oplus \lambda_a, v_b \oplus \lambda_b) = v_c \oplus \lambda_c$. We set $\chi_{i,j} = g(i, j)$.

2. The ciphertexts are the following: $\{\mathsf{Enc}_{k_{a,i}, k_{b,j}}(\chi_{i,j}, k_{c,\chi_{i,j}})\}_{i,j \in \{0,1\}}$. Then, the evaluator can decrypt the ciphertext with index $(i, j) = (v_a \oplus \lambda_a, v_b \oplus \lambda_v)$ and learn $(v_c \oplus \lambda_c, k_{c,v_c \oplus \lambda_c})$.

Finally, to let the evaluator obtain the function output, the garbler simply sends λ_w associated with the output wires to the evaluator.

The security follows from the fact that the evaluator only learns one of the two labels for each wire. This only allows him to decrypt one of the 4 ciphertexts for each gate. It is important to note that in the 2-party setting, either the garbler is corrupted or the evaluator is corrupted.

BMR Template. In the multiparty setting, we cannot let a single party act as the garbler and let all other parties act as evaluators, since the garbler may collude with some evaluator, and then security would no longer hold. The idea of the BMR construction is to let all parties jointly emulate the garbler. Note that after preparing $(\lambda_w, k_{w,0}, k_{w,1})$ for each wire, all ciphertexts can be computed in parallel. Thus, the computation task of the garbler can be represented by a constant-depth circuit, with size growing linearly in $|C|$. We may use a generic dishonest majority MPC protocol to emulate the garbler within constant rounds.

After all parties securely generate Yao's garbled circuit, each party can act as an evaluator to obtain its function output locally.

When using the state-of-the-art SPDZ-style protocol (Le Mans [RS22]) to instantiate the generic dishonest majority MPC protocol, it is possible to achieve $O(|C|n)$ communication with constant rounds to emulate the computation task of the garbler. However, an efficiency bottleneck of the BMR construction is that all parties have to emulate the underlying encryption algorithm, which needs to use the underlying symmetric-key encryption scheme in a non-black-box way. Following up works have tried to improve the concrete efficiency of the BMR construction by either making the underlying cryptographic tools used in a black-box way or using concrete instantiations for the symmetric-key encryption scheme. As we will discuss in Section 2.5, these works either require a quadratic communication in the number of parties or introduce a large multiplicative overhead.

2.2 Our Solution

Starting Point: Using Public-Key Encryption Schemes. Recall that for every wire w, we need to prepare a pair of labels $(k_{w,0}, k_{w,1})$. These labels are used as secret keys to compute proper ciphertexts. When computing the garbled circuit in a distributed way, all parties only hold shares of secret keys (wire labels) and the messages they want to encrypt. Indeed, emulating the encryption algorithm from shares of the secret key and the message is the main difficulty.

Our starting point is to replace the symmetric-key encryption scheme in the BMR template with a public-key encryption scheme. Then, each wire label becomes a key pair (pk, k) where pk is the public key and k is the private key. While all parties need to keep k private as before, pk can be made public. Looking ahead, this will help us address the difficulty of computing ciphertexts in a distributed manner.

To be more concrete, it is sufficient to address the following issues.

- For each wire w, all parties need to jointly prepare two key pairs where the private keys are additively shared among all parties.
- We need to design a protocol that allows all parties to efficiently compute a ciphertext when pk is known to all parties but the message m is additively shared.

For simplicity, we start with the semi-honest security. We will discuss how to upgrade our protocol to achieve malicious security in Section 2.4.

Addressing the First Difficulty. For the first difficulty, we note that it is sufficient to use a public-key encryption scheme that is linearly homomorphic for public keys: For two key pairs (pk_1, k_1) and (pk_2, k_2), $(\mathsf{pk}_1 \tilde{+} \mathsf{pk}_2, k_1 + k_2)$ is also a valid key pair. Here $\tilde{+}$ refers to the homomorphic operation defined by the public-key encryption scheme.

Now to generate (pk, k) such that k is additively shared among all parties,

1. Each party P_i generates (pk_i, k_i) and sends pk_i to the first party P_1;
2. P_1 locally computes $\mathsf{pk} = \mathsf{pk}_1 \tilde{+} \mathsf{pk}_2 \tilde{+} \ldots \tilde{+} \mathsf{pk}_n$ and sends pk to all parties.
3. Each party P_i views his private key k_i as an additive share of $k = k_1 + \ldots + k_n$ and outputs (pk, k_i).

Note that when there are $t = n - 1$ corrupted parties, the adversary will learn pk_i generated by the honest party P_i. However, this is fine since the adversary learning pk and public keys $\{\mathsf{pk}_j\}_{j \neq i}$ generated by all corrupted parties can anyway compute pk_i locally.

Addressing the Second Difficulty. For the second difficulty, although pk is public, the message m to be encrypted is still secret shared among all parties. It is unclear how to emulate the encryption algorithm in a black-box way.

Our main observation is that, to build a constant-round MPC protocol with linear communication, it is not necessary to obtain a single and compact ciphertext for the message m. Recall that in the evaluation phase, the evaluator will obtain the proper private key (wire label) for each wire and need to decrypt the corresponding ciphertext. We note that it is sufficient to let each party provide a separate ciphertext for his share of the message.

To be more concrete, suppose m is additively shared among all parties where each party P_i holds m_i. We let each party P_i encrypt his message share m_i using the public key pk and send the ciphertext ct_i to the evaluator. Now the evaluator with the private key k and $(\mathsf{ct}_1, \ldots, \mathsf{ct}_n)$ can already decrypt m_i from each ct_i and compute $m = m_1 + \ldots + m_n$. In this way, we can maintain linear communication while using the underlying public-key encryption scheme in a black-box way.

Summary of Our Solution. In summary, we will replace the symmetric-key encryption scheme in the BMR template by a public-key encryption scheme that is linearly homomorphic for public keys. When preparing the wire label, we let all parties prepare (pk, k) where pk is public and k is secretly shared among all parties. Then we follow the BMR template and compute the message m (which is the proper wire label (private key) for the next layer) to be encrypted, which is also secretly shared among all parties. We let each party locally encrypt his message share using pk and send the ciphertext to P_1. In the evaluation phase, P_1 will serve as the evaluator to compute and distribute the final output.

2.3 Concrete Instantiation of PKE

In our work, we instantiate the public-key encryption scheme by a variant of the well-known El-Gamal encryption scheme [Elg85] based on the Decisional Diffie-Hellman (DDH) assumption. For a DDH group G with group generator g and order p, the key generation algorithm outputs a random value $k \in \{0, \ldots, p-1\}$ as the private key and $\mathsf{pk} = g^k$ as the public key. To encrypt a group element m, the encryption algorithm samples a random value $r \in \{0, \ldots, p-1\}$ and outputs $(g^r, \mathsf{pk}^r \cdot m)$. The security follows from the DDH assumption which states that when k and r are uniformly random, given $\mathsf{pk} = g^k$ and g^r, $\mathsf{pk}^r = g^{k \cdot r}$ is computationally indistinguishable from a random group element. Thus, pk^r serves as a random mask for m.

Firstly, note that the El-Gamal encryption scheme is already linearly homomorphic for public keys: for any two key pairs (pk_1, k_1) and (pk_2, k_2), $(\mathsf{pk}_1 \cdot \mathsf{pk}_2, k_1 + k_2)$ is also a valid key pair. However, the problem with using the El-Gamal encryption scheme is that it can only be used to encrypt a group element. On the other hand, for each gate in Yao's garbled circuit, the message m that we want to encrypt is a wire label of the output wire. Recall that the wire label corresponds to the private key of the public-key encryption scheme, which is within $\{0, \ldots, p-1\}$. Taking a closer look at this issue, although pk^r is indistinguishable from a random group element, we cannot view pk^r as a uniform string due to the algebraic structure of the DDH group.

We resolve this issue by converting pk^r to a random string relying on pseudorandom generator (PRG) so that it can be used as a one-time pad key to encrypt the message m (which is also viewed as a bit-string).

- Let Ext be a strong-seeded randomness extractor Ext and Prg be a pseudorandom generator Prg. We modify the encryption algorithm as follows: After computing pk^r, we apply Ext on pk^r to obtain a (pseudo)random output. Then we apply Prg to stretch the length of the random output and use the result to encrypt m. Thus, the ciphertext is defined by $(g^r, \mathsf{seed}, m \oplus \mathsf{Prg}(\mathsf{Ext}(\mathsf{pk}^r; \mathsf{seed})))$. In practice, one can replace $\mathsf{Prg}(\mathsf{Ext}(\cdot))$ by a random oracle for practical efficiency.

We note that, in the actual construction of Yao's garbled circuit, for each gate, each message needs to be encrypted under two public keys (one from each input wire). While a direct solution is to do the encryption using the two public keys one by one, in Section 4, we provide an optimized version that directly works for the two-public-key setting.

2.4 Towards Malicious Security

So far, we have mainly focused on semi-honest security. To achieve malicious security, we rely on the standard technique of message authentication codes (MAC) [DPSZ12, DKL+13]. At the beginning of the protocol, all parties together hold an additive sharing of a global MAC key Δ, denoted by $[\Delta]$. A SPDZ sharing of a secret x is defined by a tuple of three additive sharings: $[\![x]\!] = ([x], [\Delta], [\Delta \cdot x])$. When the secret x is reconstructed, all parties can use a MAC check protocol [DKL+13] to verify the correctness of the reconstruction. We rely on the malicious variant of Le Mans [RS22] to support addition and multiplication operations over SPDZ sharings with linear communication complexity.

However, we also need to protect against the following malicious behaviors:

- Recall that to prepare a key pair (pk, k), all parties first prepare a SPDZ sharing $[\![k]\!]$. Then each party sends g^{k_i} to P_1, which allows P_1 to compute g^k and send it to all parties.

  However, parties may end up with an incorrect public key pk' either due to a corrupted party P_i sending an incorrect g^{k_i} to P_1 or because P_1 is corrupted. In the worst case, the adversary may even learn the private key (the discrete log) of pk', and the security of the public-key encryption scheme is gone.

- When doing the encryption, a corrupted party may not encrypt his correct share of the message. Then in the evaluation phase, the evaluator may decrypt an incorrect message and obtain incorrect function outputs.

- When the evaluator is corrupted, he may not send the correct function outputs to all parties at the end of the protocol.

Handling the First Attack. For the first attack, all parties will together verify the correctness of public keys. Since $(\mathsf{pk}, [\![k]\!])$ is linearly homomorphic, we simply check a random linear combination of all key pairs.

More precisely, suppose all parties have prepared $\{(\mathsf{pk}^{(i)}, [\![k^{(i)}]\!])\}_{i=0}^{N}$. To verify the correctness of each $\mathsf{pk}^{(i)}$, all parties compute a random linear combination

$$(\mathsf{pk}, [\![k]\!]) = (\mathsf{pk}^{(0)}, [\![k^{(0)}]\!]) + \sum_{i=1}^{N} r_i \cdot (\mathsf{pk}^{(i)}, [\![k^{(i)}]\!]),$$

where $r_1, \ldots, r_N$ are (pseudo)random coefficients. Now it is sufficient to check whether pk is correct with respect to k. Relying on the MAC, all parties can verifiably reconstruct the secret k and then check whether $\mathsf{pk} = g^k$.

Handling the Last Two Attacks. We note that given a key pair (pk, k), one can verify whether it is valid by checking $\mathsf{pk} = g^k$. Since the public keys are known to all parties, an honest party can use this property to verify the correctness of a wire label (private key).

The second attack only occurs when the evaluator is honest. In the evaluation phase, whenever an honest evaluator computes a wire label, he can use the above approach to check the validity of the wire label. In case he does not obtain the correct wire label, the protocol will abort. In this way, the protocol will only proceed if an honest evaluator obtains the correct wire label for each wire.

For the third attack, we require the evaluator to send the wire label for the output wires to all parties so that an honest receiver can check the validity of the wire label locally. Note that a malicious evaluator can only learn the wire label corresponding to the function output. In this way, an honest receiver will not accept an incorrect function output.

Optimizations. We note that in the evaluation phase, for each wire, the evaluator will only learn one of the two wire labels associated with this wire. Thus, we may set the difference between these two wire labels to be the same for all wires. This trick has been used in many previous works for constructing efficient Yao's garbled circuits.

More concretely, for each wire w, recall that all parties need to prepare $(\mathsf{pk}_{w,0}, [k_{w,0}])$ and $(\mathsf{pk}_{w,1}, [k_{w,1}])$. We require that $k_{w,1} - k_{w,0} = \Delta$, where Δ is the MAC key of the underlying SPDZ protocol. This brings us two benefits.

- First, when computing $\mathsf{pk}_{w,0}$ and $\mathsf{pk}_{w,1}$, it is sufficient to compute g^{Δ} and $\mathsf{pk}_{w,0}$. Then all parties can locally compute $\mathsf{pk}_{w,1} = g^{\Delta} \cdot \mathsf{pk}_{w,0}$. In this way, we only need to prepare one key pair for each wire.
- Second, in the Yao's garbled circuit, recall that for each gate, we need to compute 4 ciphertexts. Let a, b denote the input wires of this gate and c denote the output wire. For all $i, j \in \{0, 1\}$, we need to first compute $\chi_{i,j}$, which is the index of the private key we need to encrypt under the public keys $\mathsf{pk}_{a,i}$ and $\mathsf{pk}_{b,j}$; i.e., the ciphertext we need to compute is $\mathsf{Enc}_{\mathsf{pk}_{a,i},\mathsf{pk}_{b,j}}(\chi_{i,j}, k_{c,\chi_{i,j}})$[1]. Then we need to compute an additive sharing of $k_{c,\chi_{i,j}}$. Usually, this is done by first computing a SPDZ sharing $[\![\chi_{i,j}]\!]$ and then using $\chi_{i,j}$ to choose one of the two private keys, which requires one additional multiplication operation.

  We observe that $k_{c,\chi_{i,j}} = k_{c,0} + \chi_{i,j} \cdot \Delta$. Note that all parties have already held $[\chi_{i,j} \cdot \Delta]$ from $[\![\chi_{i,j}]\!]$. Thus, all parties can locally compute $[k_{c,\chi_{i,j}}] = [k_{c,0}] + [\chi_{i,j} \cdot \Delta]$.

Remark 1 (Free XOR and the Assumption of Random Oracle)**.** *In our construction, we have to use a large prime field due to the DDH assumption. This unfortunately is not compatible with the free-XOR technique introduced in [PS08], which requires working over an extension field of the binary field. We leave the question of incorporating free-XOR into our technique to future work.*

As in all previous works that use the same difference between the two labels for all wires, this optimization only works under the assumption of a random oracle due to the issue of circular encryption. For concrete efficiency, we will mainly focus on the construction with the above optimization assuming a random oracle and refer the readers to Appendix G for the one that does not require a random oracle.

2.5 Related Works

As we have mentioned above, the main efficiency bottleneck of the BMR construction is that it requires all parties to emulate the underlying encryption algorithm, which needs to use the underlying symmetric-key encryption scheme in a non-black-box way. For typical instantiations, the symmetric-key encryption scheme involves computation of a pseudo-random function (PRF). To be more concrete, to encrypt a message m with secret key k for a gate with identifier g, the ciphertext is defined by

$$\mathsf{ct} = \mathsf{PRF}_k(g) \oplus m.$$

Starting from secret sharings of k and m, the joint computation of the PRF would incur a large overhead in both communication and computation depending upon the circuit size of the PRF.

To overcome this issue, Damgård and Ishai [DI05] proposed a variant of the BMR construction that uses PRG in a black-box way. At a high level, instead of viewing that all parties hold a secret sharing for each wire label, we simply take the concatenation of all shares as the wire label $k = k_1 \| k_2 \| \ldots \| k_n$. Now the ciphertext is defined by

$$\mathsf{ct} = \mathsf{PRF}_{k_1}(g) \oplus \mathsf{PRF}_{k_2}(g) \oplus \ldots \oplus \mathsf{PRF}_{k_n}(g) \oplus m.$$

In this way, each party can locally apply PRF on his share and only secret share the result. However, since the size of each wire label is increased by a factor of n because of concatenation, both the size of the garbled circuit and the overall communication complexity are increased by a factor of n, i.e., $O(|C|n)$ for the size of the garbled circuit with $O(|C|n^2)$ communication!

A line of works focuses on partially resolving this issue by either considering a weaker security where there are more than 1 honest parties [HOSS18a, HOSS18b, BGH+23] or only reducing the size of the garbled circuit relying on advanced cryptographic tools and assumptions [BLO17, BCO+21]. We note that [BLO17] is also based on the DDH assumption and [BCO+21] has the potential of achieving overall linear communication complexity, which we will elaborate on below.

[1] In our actual construction, we only encrypt $k_{c,\chi_{i,j}}$ but not $\chi_{i,j}$ since the evaluator can learn $\chi_{i,j}$ by comparing $k_{c,\chi_{i,j}}$ with the two public keys $\mathsf{pk}_{c,0}, \mathsf{pk}_{c,1}$.

Comparison with [BLO17]. The protocol in [BLO17] follows the BMR framework to encrypt secret-shared messages with secret-shared keys. By using key-homomorphic PRF, the size of the garbled circuit is independent of the number of parties. However when using DDH, the key-homomorphic PRF is multiplicative homomorphic which requires them to multiply n values, one held by each party. This step incurs a quadratic communication. Besides, [BLO17] only works against semi-honest adversaries.

Our work uses PKE and sacrifices the succinctness of the garbled circuit (i.e., the size is linear in the number of parties) to achieve linear communication.

Achieving Linear Communication From [BCO+21]. To reduce the size of the garbled circuit, the work [BCO+21] relies on a symmetric-key encryption scheme based on the LPN assumption. At a high level, to encrypt a message m with secret key k, the encryption algorithm is defined by

$$\mathsf{ct} = (\boldsymbol{A} \cdot k) \oplus e \oplus (\boldsymbol{M} \cdot m),$$

where $\boldsymbol{A}$ and $\boldsymbol{M}$ are public matrices, k, m are viewed as vectors of bits, and e is a bit-string (or a vector of bits) with each bit independently drawn from some Bernoulli distribution. Intuitively, the LPN assumption states that for a random string k, given $\boldsymbol{A}$, $(\boldsymbol{A} \cdot k) \oplus e$ is computationally indistinguishable from a uniform string, which is used as the one-time pad key to encrypt the message. To decrypt ct with k, one can compute $\mathsf{ct} \oplus (\boldsymbol{A} \cdot k)$ which is equal to $e \oplus (\boldsymbol{M} \cdot m)$. The matrix $\boldsymbol{M}$ is used to encode the message m so that the message can be recovered even if some bits of the codeword are incorrect due to the error string e. Note that the key size and the ciphertext size do not grow with the number of parties, thus achieving $O(|C|)$ for the size of the garbled circuit.

The main benefit of this encryption scheme is that, if parties have additive sharings of k and m, and can generate additive sharings of e, then the encryption algorithm can be computed via local computation. In [BCO+21], the generation of additive sharings of error strings and the computation of secret sharings of k and m are done via the SPDZ protocol in a black-box way. When instantiating the underlying SPDZ protocol by Le Mans [RS22], it is possible to achieve linear communication $O(|C|n)$[2].

Despite the potential of achieving linear communication, the LPN assumption usually requires the use of a very large parameter to achieve the desired level of security. For example, the analysis in [BCO+21] shows that when the computational security parameter $\lambda = 128$ and the statistical security parameter $\kappa = 80$, the ciphertext size needs to be $\ell = 8925$ bits. Furthermore, all parties need to jointly prepare ℓ random bits that follow the Bernoulli distribution to obtain e for each encryption. In [BCO+21], to prepare one random bit sharing such that the secret is 1 with probability τ, all parties first generate $\log(1/\tau)$ random bit sharings where the secrets are uniformly distributed and then multiply them together. As a result, there is a large constant overhead in [BCO+21].

Implicit Overhead of LPN-based Encryption. We note the symmetric-key encryption scheme used in [BCO+21] satisfies an even stronger homomorphism: Each party can locally encrypt his message share using his key share. Then the summation of all ciphertexts corresponds to a valid ciphertext of m under the secret key k. This seems to give a way to avoid generating the error string e jointly. Instead, each party P_i can generate his own error string e_i (used to encrypt his message share) and the final error string becomes $e = \bigoplus_{i=1}^{n} e_i$.

However, even if all parties perform honestly, note that the error accumulates in the LPN-based encryption scheme and this requires the ciphertext to be long enough so that one can still recover the message during the decryption phase. This would implicitly require the ciphertext size to be linear in n. This is why in both [BCO+21, BGH+23], the error string is generated jointly.

A similar issue would occur if one tries to instantiate the public-key encryption scheme used in our construction from the LPN assumption: The error accumulated during the aggregation of all parties' public keys

[2]We note that [BCO+21] actually needs to compute binary circuits using the SPDZ protocol. However, the malicious variant of Le Mans [RS22] currently only works for a large finite field. In this work, we omit this distinction when comparing [BCO+21] with our result.

may require the key size to be proportional to the number of parties, resulting in a quadratic communication overhead.

Comparison with [GYKW24]. We note that a recent work [GYKW24] also achieves linear communication for computing the garbled circuit and the garbled circuit size is independent of the number of parties. However, their construction only focuses on semi-honest security and requires a threshold homomorphic encryption scheme, while our protocol achieves malicious security with lighter assumptions. In addition, their construction requires $O(\log n)$ rounds to compute the garbled circuits.

3 Preliminaries

Notations. Let $\mathbb{F}_p$ be a prime field of order p. Let U_m be the uniform distribution over $\{0,1\}^m$. We use κ to denote the statistical security parameter, and we use λ to denote the computational security parameter.

3.1 Basic Definitions and Primitives

We first present some basic definitions.

Definition 1. (Min-entropy). *Let X be a random variable. Then the min-entropy of X is defined as*

$$H_\infty(X) = \min_x \log \frac{1}{\Pr[X = x]}.$$

Definition 2. (k-source). *A random variable X is a k-source if $H_\infty(X) \geq k$.*

Definition 3. (Statistical Distance). *For random variables X and Y taking values in $\mathcal{X}$, their statistical difference is defined as*

$$\Delta(X,Y) = \max_{T \subset \mathcal{X}} |\Pr[X \in T] - \Pr[Y \in T]|.$$

We say that X and Y are ϵ-close if $\Delta(X,Y) \leq \epsilon$.

An extractor [NZ96] can be used to extract uniform randomness out of a weakly random value which is only assumed to have sufficient min-entropy. A strong seeded extractor is defined as follows.

Definition 4. (Strong Seeded Extractors [NZ96]). *An efficient function $\mathsf{Ext} : \mathcal{X} \times \{0,1\}^d \to \{0,1\}^m$ is a strong (k,ϵ)-extractor if for every k-source X on $\mathcal{X}$, $(U_d, \mathsf{Ext}(X, U_d))$ is ϵ-close to (U_d, U_m).*

A strong seeded extractor can be constructed with an efficient universal hash function [CW79, Sti91, NP99, PS08], which follows from the Leftover Hash Lemma [HILL99]. We state this result below.

Theorem 2. *There exists a strong $(k, 2^{-\kappa})$-seeded extractor $\mathsf{Ext} : \mathcal{X} \times \{0,1\}^k \to \{0,1\}^{k-2\kappa}$.*

We also need a pseudorandom generator for our encryption scheme.

Definition 5. (Computational Distance). *For random variables X and Y taking values in $\mathcal{X}$, the advantage of a circuit D in distinguishing X and Y is defined as*

$$\Delta_D(X,Y) = |\Pr[D(X) = 1] - \Pr[D(Y) = 1]|.$$

Let $\mathcal{D}_t$ be the set of all probabilistic circuits of size t. The computational difference of X and Y is defined as

$$\mathsf{CD}_t(X,Y) = \max_{D \in \mathcal{D}_t} \Delta_D(X,Y).$$

When $X = X_\lambda$ and $Y = Y_\lambda$ are families of distributions indexed by a security parameter λ, we say that X and Y are computationally indistinguishable, denoted $X =_c Y$, if for every polynomial $t(\cdot)$, $\mathsf{CD}_{t(\lambda)}(X,Y) = \mathsf{negl}(\lambda)$.

Definition 6. (Pseudorandom Generator). *A length-increasing function* $\mathsf{Prg} : \{0,1\}^\lambda \to \{0,1\}^m$ *is a pseudorandom generator (PRG) if* $\mathsf{Prg}(U_\lambda) =_c U_m$.

Decisional Diffie-Hellman (DDH) Assumption. We assume that the DDH assumption holds, i.e., for a group G of a 2λ-bit prime order p, let g be a generator of G, then:

$$\{g^a, g^b, g^{ab} : a, b \xleftarrow{\$} \mathbb{F}_p\} =_c \{g^a, g^b, g^c : a, b, c \xleftarrow{\$} \mathbb{F}_p\}.$$

Security Model. We define the security of multiparty computation in the *real and ideal world paradigm* [Can00]. Informally, 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. For more details, we refer the readers to Appendix A.

3.2 Secret Sharing

Let $[x]$ denote an unauthenticated additive sharing of x with P_i's share $x^{(i)}$. We use $\mathcal{P} = \{P_1, \ldots, P_n\}$ to denote the set of all parties. For all $\mathcal{P}_A, \mathcal{P}_B \subset \mathcal{P}$, we follow the definition in [RS22] to define an authenticated sharing $\langle x \rangle^{\mathcal{P}_A, \mathcal{P}_B}$ as follows.

- All parties in $\mathcal{P}_A$ together hold an additive sharing of x. Each party $P_j \in \mathcal{P}_B$ holds a global key $\Delta^{(j)}$.
- For every $P_i \in \mathcal{P}_A, P_j \in \mathcal{P}_B$, P_j holds a random local key $K_i^{(j)}$ and P_i holds the MAC of $x^{(i)}$ defined by
$$M_j^{(i)} = K_i^{(j)} + \Delta^{(j)} \cdot x^{(i)}.$$

An authenticated additive sharing

$$\langle x \rangle^{\mathcal{P}_A, \mathcal{P}_B} = ((x^{(i)}, (M_j^{(i)})_{P_j \in \mathcal{P}_B})_{P_i \in \mathcal{P}_A}, (\Delta^{(j)}, (K_i^{(j)})_{P_i \in \mathcal{P}_A})_{P_j \in \mathcal{P}_B})$$

can be locally converted to a SPDZ sharing $[\![x]\!] = ([x], [\Delta], [\Delta \cdot x])$ when $\mathcal{P}_B = \mathcal{P}$ by letting each P_i compute

$$\left(x^{(i)}, \Delta^{(i)}, \Delta^{(i)} \cdot x^{(i)} + \sum_{j \neq i} (M_j^{(i)} - K_j^{(i)}) \right),$$

where $x^{(i)} = M_j^{(i)} = K_i^{(j)} = 0$ for all $P_i \notin \mathcal{P}_A$ and $P_j \in \mathcal{P}$. When $\mathcal{P}_A = \mathcal{P}_B = \mathcal{P}$, we simply use $\langle x \rangle$ to denote $\langle x \rangle^{\mathcal{P}, \mathcal{P}}$.

3.3 Functionalities for Sub-protocols

We borrow the following functionalities from [RS22].

Programmable OLE. We use a functionality for *random, programmable oblivious linear evaluation* (OLE), $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ (see Figure 1). This is a two-party functionality, which computes a batch of secret-shared products, i.e. random tuples $(u_i, v_i), (x_i, w_i)$, where $w_i = u_i \cdot x_i + v_i$, over the field $\mathbb{F}_p$. The *programmability* requirement is that, for any given instance of the functionality, the party who obtains u_i or x_i can program these to be derived from a chosen random seed. This allows the same u_i, v_i to be used in different instances of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. We model the programmability with an expansion function Expand, which is a PRG.

Functionality $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$

Let $\mathsf{Expand} : S \to \mathbb{F}_p^m$ be an expansion function with seed space S and output length m. On receiving $s_a \in S$ from P_A and $s_b \in S$ from P_B:

1. The trusted party computes $\boldsymbol{u} = \mathsf{Expand}(s_a), \boldsymbol{x} = \mathsf{Expand}(s_b)$ and samples $\boldsymbol{v} \in \mathbb{F}_p^m$.
2. The trusted party outputs $\boldsymbol{w} = \boldsymbol{u} * \boldsymbol{x} + \boldsymbol{v}$ to P_A and $\boldsymbol{v}$ to P_B.

Corrupted Parties: If P_B is corrupt, $\boldsymbol{v}$ may be chosen by $\mathcal{S}$. For a corrupt P_A, $\mathcal{S}$ can choose $\boldsymbol{w}$ (and then $\boldsymbol{v}$ is recomputed accordingly).

Figure 1: Functionality for programmable OLE.

Multiparty VOLE. *Vector oblivious linear evaluation* (VOLE) can be seen as a batch of OLEs with the same x_i value in each tuple, i.e., a vector $\boldsymbol{w} = \boldsymbol{u} \cdot x + \boldsymbol{v}$, where $x \in \mathbb{F}_p$ is a scalar given to one party. In the functionality of multiparty VOLE, $\mathcal{F}_{\mathsf{nVOLE}}$ (see Figure 2), every pair of parties (P_i, P_j) is given a random VOLE instance $\boldsymbol{w}_j^{(i)} = \boldsymbol{u}^{(i)} \cdot x^{(j)} + \boldsymbol{v}_i^{(j)}$. The functionality guarantees consistency, in the sense that the same $\boldsymbol{u}^{(i)}$ or $x^{(j)}$ values will be used in each instance involving P_i or P_j. While, unlike the OLE functionality, the $\boldsymbol{u}^{(i)}, x^{(i)}$ values in $\mathcal{F}_{\mathsf{nVOLE}}$ are not programmable, we do require that the functionality outputs to P_i a short seed which can be expanded to $\boldsymbol{u}^{(i)}$ so that P_i can later use this as an input to $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$.

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

Let $\mathsf{Expand} : S \to \mathbb{F}_p^m$ be an expansion function with seed space S and output length m.

Init: On receiving Init from P_i for $i \in [1, n]$, the trusted party samples $\Delta^{(i)} \in \mathbb{F}_p$, sends it to P_i, and ignores all subsequent Init commands from P_i.

Extend: On receiving Extend from every $P_i \in \mathcal{P}$:

1. For each honest P_i, the trusted party randomly samples $\mathsf{seed}^{(i)} \in S$.
2. For each P_i, let $\boldsymbol{u}^{(i)} = \mathsf{Expand}(\mathsf{seed}^{(i)})$. The trusted party samples $(\boldsymbol{v}_i^{(j)})_{j \neq i} \in \mathbb{F}_p^m$, retrieves $\Delta^{(j)}$ and computes $\boldsymbol{w}_j^{(i)} = \boldsymbol{u}^{(i)} \cdot \Delta^{(j)} + \boldsymbol{v}_i^{(j)}$.
3. If P_j is corrupt, $\mathcal{S}$ can send a set I to the trusted party. If $\mathsf{seed}^{(i)} \in I$, the trusted party sends $\mathsf{success}$ to P_j and continues. Otherwise, the trusted party sends abort to both parties, outputs $\mathsf{seed}^{(i)}$ to P_j and aborts.
4. The trusted party outputs $((\mathsf{seed}^{(i)}, (\boldsymbol{w}_j^{(i)}, \boldsymbol{v}_j^{(i)})_{j \neq i})$ to P_i.

Corrupted Parties: A corrupt P_i can choose $\Delta^{(i)}$ and $\mathsf{seed}^{(i)}$. It can also choose $\boldsymbol{w}_j^{(i)}$ (and then $\boldsymbol{v}_i^{(j)}$ is recomputed accordingly) and $\boldsymbol{v}_j^{(i)}$.

Global key query: If P_i is corrupted, the trusted party receives $(\mathsf{guess}, \boldsymbol{\Delta}')$ from $\mathcal{S}$ with $\boldsymbol{\Delta}' \in \mathbb{F}_p^n$. If $\boldsymbol{\Delta}' = \boldsymbol{\Delta}$ where $\boldsymbol{\Delta}' = (\Delta^{(1)}, \ldots, \Delta^{(n)})$, the trusted party sends $\mathsf{success}$ to P_i and ignores any subsequent global key query, otherwise, the trusted party sends $(\mathsf{abort}, \boldsymbol{\Delta})$ to P_i, abort to P_j and aborts.

Figure 2: Functionality for n-party VOLE.

We also use the standard functionalities $\mathcal{F}_{\mathsf{Coin}}$ (Figure 3) and $\mathcal{F}_{\mathsf{Commit}}$ (Figure 4) for generating random coins and commitments, respectively.

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

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

Figure 3: Functionality for generating a common coin.

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

Commit: On input $(\mathsf{commit}, P_i, x, \tau_x)$ from P_i, where τ_x is a previously unused identifier, the trusted party stores (P_i, x, τ_x) and sends (P_i, τ_x) to all parties.

Open: On input $(\mathsf{open}, P_i, \tau_x)$ from P_i, the trusted party retrieves x and sends (x, i, τ_x) to all the parties.

Figure 4: Functionality for commitment.

3.4 MAC Check on Opened Values.

For a SPDZ sharing $[\![x]\!]$, when we say the parties open $[\![x]\!]$, we mean that the parties run the following Π_{Open} protocol on $[\![x]\!]$.

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

1. All the parties send their shares of $[x]$ to P_1.
2. P_1 reconstructs x and sends it to all the parties.

Figure 5: Protocol to open a SPDZ sharing.

When the secrets of some SPDZ sharings are opened to all parties, they can check the correctness relying on the MACs by 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 6).

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

Let all parties agree on a PRG $\mathsf{Prg} : \mathbb{F}_p \to \mathbb{F}_p^m$. 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 seed in $\mathbb{F}_p$ and expand it with Prg to get random random values $\chi_1, \ldots, \chi_m \in \mathbb{F}_p$.
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 $(\mathsf{commit}, P_i, [\sigma], \tau_{[\sigma]})$.
4. The parties open their commitments and check that $\sum_{i=1}^n [\sigma] = 0$. If not, the parties output abort and abort the protocol.

Figure 6: Protocol for MAC checking.

4 Encryption Scheme Based on DDH

In this section, we construct a public-key encryption scheme based on the DDH assumption. We will first give a construction based on strong seeded extractors, and then simplify it under the assumption of random oracles.

4.1 Encryption Scheme Based on Strong Seeded Extractors

We now introduce our construction of an encryption scheme based on strong seeded extractors. Let p be a prime number and $\mathbb{F}_p$ be the prime field of size p. Our goal is to encrypt a message in $\mathbb{F}_p$.

Let $\oplus$ denote the bit-wise XOR of two binary strings of the same length. Let $k = \max\{2\lambda, \lambda + 2\kappa\}$. Our encryption scheme PKE_1 is a tuple of four PPT algorithms $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ defined as follows:

- $\mathsf{Setup}(\lambda, \kappa)$: The setup algorithm Setup samples the following:

  1. A group G of order p with a generator g, where p is a k-bit prime. Let ℓ denote the length of group elements in G.
  2. A strong $(k, 2^{-\kappa})$-extractor $\mathsf{Ext} : \{0,1\}^\ell \times \{0,1\}^k \to \{0,1\}^\lambda$ (Theorem 2).
  3. A pseudorandom generator $\mathsf{Prg} : \{0,1\}^\lambda \to \{0,1\}^k$.

  Setup outputs public parameters $pp = (G, p, g, \mathsf{Ext}, \mathsf{Prg}, \lambda, \kappa)$.

- $\mathsf{Gen}(pp)$: The key-generation algorithm Gen samples $\mathsf{sk}_1, \mathsf{sk}_2 \in \mathbb{F}_p$ and computes $\mathsf{pk}_1 = g^{\mathsf{sk}_1}$, $\mathsf{pk}_2 = g^{\mathsf{sk}_2}$. Gen outputs $(\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2)$, where $\mathsf{sk}_1, \mathsf{sk}_2$ are the secret keys, and $\mathsf{pk}_1, \mathsf{pk}_2$ are the public keys.

- $\mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m)$: The encryption algorithm Enc runs as follows:
  1. Sample random $k_1, k_2 \in \mathbb{F}_p$, and compute $g^{k_1}, g^{k_2}, (\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2}$.
  2. Sample random $s \in \{0,1\}^k$, compute $m' = \mathsf{Prg}(\mathsf{Ext}((\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2}, s))$.
  3. Encode m to a vector in $\{0,1\}^k$. Output $c = (m \oplus m', s, g^{k_1}, g^{k_2})$.
- $\mathsf{Dec}(pp, \mathsf{sk}_1, \mathsf{sk}_2, c)$: Suppose $c = (m^*, s, g_1, g_2)$. Then, the decryption algorithm Dec outputs $m = m^* \oplus \mathsf{Prg}(\mathsf{Ext}(g_1^{\mathsf{sk}_1} \cdot g_2^{\mathsf{sk}_2}, s))$.

This public key encryption scheme guarantees that a party can decrypt a ciphertext with both secret keys, but he does not learn any information about the message with only one key. We state this as the following theorem.

Theorem 3. $\mathsf{PKE}_1 = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *satisfies the following conditions:*

- **Correctness.** *Let* $pp \leftarrow \mathsf{Setup}(\lambda, \kappa)$. *For any message* $m \in \mathbb{F}_p$,

$$\Pr\left[\mathsf{Dec}(pp, \mathsf{sk}_1, \mathsf{sk}_2, c) = m : \begin{array}{l}(\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp), \\ c \leftarrow \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m)\end{array}\right] = 1.$$

- **Security.** *Let* $pp \leftarrow \mathsf{Setup}(\lambda, \kappa)$. *Assume the DDH assumption over* G *with group generator* g. *Then for any pair of messages* $m_0, m_1 \in \mathbb{F}_p$,

$$\begin{aligned}&\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ =_c\ &\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ &\qquad\qquad \textit{and} \\ &\{\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ =_c\ &\{\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}.\end{aligned}$$

We give the proof of this theorem in Appendix B.1.

Instantiation of the Group. In practice, we can choose G to be an elliptic curve $E(\mathbb{F}_q)$ of size p. Each point on this elliptic curve can be expressed as a point in $\mathbb{F}_q^2$. Thus, we can take $\ell = 2\log q$, where it is ensured by the Mordell-Weil Theorem (see [Mor22, Wei29]) that $\big||E(\mathbb{F}_q)| - q - 1\big| < 2\sqrt{q}$, which means $q = O(p)$.

4.2 Encryption Scheme Based on Random Oracle

If we assume the existence of a random oracle, the encryption scheme can be much simpler. We provide an encryption scheme $\mathsf{PKE}_2 = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$:

- $\mathsf{Setup}(\lambda)$: The setup algorithm Setup samples a group G of order p with a generator g, where p is a 2λ-bit prime. Then, Setup initializes a random oracle $\mathcal{O}$ with output length 2λ. Setup outputs $pp = (G, p, g, \mathcal{O}, \lambda)$.
- $\mathsf{Gen}(pp)$: The key-generation algorithm Gen samples $\mathsf{sk}_1, \mathsf{sk}_2 \in \mathbb{F}_p$ and computes $\mathsf{pk}_1 = g^{\mathsf{sk}_1}$, $\mathsf{pk}_2 = g^{\mathsf{sk}_2}$. Gen outputs $(\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2)$, where $\mathsf{sk}_1, \mathsf{sk}_2$ are secret keys, and $\mathsf{pk}_1, \mathsf{pk}_2$ are public keys.
- $\mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m)$: The encryption algorithm Enc runs as follows:
  1. Sample random $k_1, k_2 \in \mathbb{F}_p$, then compute g^{k_1}, g^{k_2}, and $(\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2}$.
  2. Query the random oracle $\mathcal{O}$ with input $(\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2}$ to obtain $m' = \mathcal{O}((\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2})$.
  3. Encode m to a vector in $\{0,1\}^{2\lambda}$. Output $c = (m \oplus m', g^{k_1}, g^{k_2})$.

- $\mathsf{Dec}(pp, \mathsf{sk}_1, \mathsf{sk}_2, c)$: Suppose $c = (m^*, g_1, g_2)$. Then, the decryption algorithm Dec outputs $m = m^* \oplus \mathcal{O}(g_1^{\mathsf{sk}_1} \cdot g_2^{\mathsf{sk}_2})$.

Similarly to PKE_1, PKE_2 satisfies the correctness and security properties.

Theorem 4. $\mathsf{PKE}_2 = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *satisfies the following conditions:*

- **Correctness.** *Let* $pp \leftarrow \mathsf{Setup}(\lambda)$. *For any message* $m \in \mathbb{F}_p$,

$$\Pr\left[\mathsf{Dec}(pp, \mathsf{sk}_1, \mathsf{sk}_2, c) = m : \begin{array}{l}(\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp), \\ c \leftarrow \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m)\end{array}\right] = 1.$$

- **Security.** *Let* $pp \leftarrow \mathsf{Setup}(\lambda)$. *Assume the DDH assumption over* G *with group generator* g. *For any pair of messages* $m_0, m_1 \in \mathbb{F}_p$,

$$\begin{aligned} &\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ =_c\ &\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ &\qquad\qquad \textit{and} \\ &\{\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ =_c\ &\{\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}. \end{aligned}$$

We give the proof of this theorem in Appendix B.2.

5 Preprocessing Phase

5.1 Preprocessing Functionality

In the preprocessing phase, all the parties need to prepare several sharings to be used in the main protocol. The preprocessing functionality $\mathcal{F}_{\mathsf{prep}}$ is defined in Figure 7. It allows the parties to prepare the following sharings:

- **Random Values:** A SPDZ sharing $[\![r]\!]$ of a random element $r \in \mathbb{F}_p$.
- **Triples:** SPDZ sharings $[\![a]\!], [\![b]\!], [\![c]\!]$ for $c = a \cdot b$, where a, b are random elements in $\mathbb{F}_p$.
- **Random Bits:** SPDZ sharings $[\![\lambda]\!]$, where $\lambda \in \{0, 1\} \subset \mathbb{F}_p$ is a random bit.

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

Init: On receiving $(\mathsf{Init}, m_T, m_R, m_B)$ from every P_i, the trusted party samples a random element in $\mathbb{F}_p$ as Δ and sends g^{Δ} to $\mathcal{S}$. Then:

1. For each corrupted party P_i, the trusted party receives $\Delta^{(i)}$ from $\mathcal{S}$ and sends it to P_i.
2. The trusted party samples a MAC key $\Delta^{(i)} \leftarrow \mathbb{F}_p$ for each honest party P_i such that $\sum_{i=1}^{n} \Delta^{(i)} = \Delta$. Then, it sends $\Delta^{(i)}$ to each honest party P_i.

Finally, the trusted party sends g^{Δ} to all the parties and ignores subsequent Init commands from each P_i.

Random Values: Repeat the following m_R times:

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

Triples: Repeat the following m_T times:

1. Run steps 1 and 2 from **Random Values** twice, to create sharings $[\![a]\!], [\![b]\!]$.

2. Let $c = a \cdot b$. The trusted party computes $\Delta \cdot c$.
3. The trusted party receives corrupted parties' shares of $[\![c]\!]$ from $\mathcal{S}$ and randomly samples honest parties' shares of $[\![c]\!]$ based on the secret and corrupted parties' shares.
4. The trusted party outputs $([\![a]\!], [\![b]\!], [\![c]\!])$ to the parties.

Random Bits: Repeat the following m_B times:

1. The trusted party randomly samples $\lambda \in \{0, 1\}$.
2. The trusted party receives corrupted parties' shares of $[\![\lambda]\!]$ from $\mathcal{S}$ and randomly samples honest parties' shares of $[\![\lambda]\!]$ based on the secret and corrupted parties' shares.
3. The trusted party outputs $[\![\lambda]\!]$ to the parties.

Abort. Upon receiving abort from $\mathcal{S}$, the trusted party sends Δ to $\mathcal{S}$, outputs abort to all the parties, and aborts.

Figure 7: Functionality for preprocessing.

With the authenticated triples, the parties can do multiplication following the SPDZ protocol. We present Π_{Mult} in Figure 8.

Protocol $\Pi_{\mathsf{Mult}}([\![x]\!], [\![y]\!])$

To compute $[\![z]\!] = [\![x]\!] \cdot [\![y]\!]$:

1. Using a triple $([\![a]\!], [\![b]\!], [\![c]\!])$ generated using **Triples** from $\mathcal{F}_{\mathsf{prep}}$, the parties compute and open $[\![e]\!] = [\![x - a]\!], [\![d]\!] = [\![y - b]\!]$ by Π_{Open}.
2. The parties compute the multiplication by $[\![z]\!] = e \cdot d + e \cdot [\![b]\!] + d \cdot [\![a]\!] + [\![c]\!]$.

Figure 8: Protocol for SPDZ multiplication.

5.2 Preprocessing Protocol

For simplicity, we use $\boldsymbol{x}[k]$ to denote the k-th entry of $\boldsymbol{x}$. In Figure 9, we provide our protocol Π_{prep} realizing the preprocessing functionality $\mathcal{F}_{\mathsf{prep}}$ in the $\{\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{nVOLE}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model.

Protocol Π_{prep}

Let $\mathsf{Expand} : S \to \mathbb{F}_p^m$ be an expansion function with seed space S and output length $m = \max\{m_T, m_R + 1, m_B\}$. Each call of $\mathcal{F}_{\mathsf{nVOLE}}$ or $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ is with respect to this Expand and m.
Let G be a multiplicative group of order p with generator g.

Init:

1. Each $P_i \in \mathcal{P}$ sends Init to $\mathcal{F}_{\mathsf{nVOLE}}$ and receives $\Delta^{(i)}$.
2. Each P_i computes $g^{\Delta^{(i)}}$ and sends it to all the parties.
3. Each party locally computes $g^{\Delta} = \prod_{i=1}^{n} g^{\Delta^{(i)}}$.

The following protocols are then executed in order: $\Pi_{\mathsf{rand}}, \Pi_{\mathsf{trip}}, \Pi_{\mathsf{bit}}, \Pi_{\mathsf{ver}}$.

Figure 9: Protocol for preprocessing.

Each random SPDZ sharing is obtained by conversion from an authenticated additive sharing. The authenticated additive sharings are generated by $\mathcal{F}_{\mathsf{nVOLE}}$. More concretely, the parties need to invoke $\mathcal{F}_{\mathsf{nVOLE}}$ in order to receive the seeds and pair-wise MACs from $\mathcal{F}_{\mathsf{nVOLE}}$. Then, the parties expand their own seeds to get their shares of random additive sharings. We will generate m_R random sharings together with another random sharing for verification of g^{Δ}. The protocol Π_{rand} for generating random SPDZ sharings is given in Figure 10.

Protocol Π_{rand}

Random Values:

- *Setup:* Each party P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ with input Extend. P_i receives $(s_r^{(i)}, (\boldsymbol{M}_j^{(i)}, \boldsymbol{K}_j^{(i)})_{j\neq i})$. Each P_i's $\mathsf{Expand}(s_r^{(i)}), \Delta^{(i)}, (\boldsymbol{M}_j^{(i)}, \boldsymbol{K}_j^{(i)})_{j\neq i}$ form his shares of a vector of m sharings $\langle \boldsymbol{r} \rangle$.
- To get the k-th random sharing ($1 \leq k \leq m_R + 1$), each party takes the k-th share from $\langle \boldsymbol{r} \rangle$ and locally convert it to $[\![r]\!]$.

Figure 10: Protocol for preparing random sharings.

To prepare multiplication triples, the parties first prepare two seeds for random SPDZ sharings, and then pair-wise invoke $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with their seeds, which enables each pair of parties to obtain a two-party additive sharing of the product of their shares of $[\boldsymbol{a}], [\boldsymbol{b}]$ expanded from their seeds. Adding all these shares together, the parties get unauthenticated additive sharings of $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. To convert an unauthenticated additive sharing $[c]$ to the SPDZ sharing $[\![c]\!]$, the parties need another random sharing $[\![\ell]\!]$, so that they can open $c + \ell$ and compute $[\![c]\!] = (c + \ell) - [\![\ell]\!]$. Since the corrupted parties may not send the correct seeds to $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, we need to prepare extra triples for verification. We utilize the technique from MASCOT [KOS16] to use an extra triple $[\![a']\!], [\![b]\!], [\![c']\!]$ to verify the correctness of $c = a \cdot b$ for each triple $[\![a]\!], [\![b]\!], [\![c]\!]$. The protocol Π_{trip} for triple generation is given in Figure 11.

Protocol Π_{trip}

Triples:

- *Setup:*
  1. Each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ 5 times, with input Extend, and receives the seeds $s_a^{(i)}, s_b^{(i)}, s_a^{(i)'}, s_\ell^{(i)}, s_\ell^{(i)'}$. The outputs define vectors of shares $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle, \langle \boldsymbol{a}' \rangle, \langle \boldsymbol{\ell} \rangle, \langle \boldsymbol{\ell}' \rangle$ such that $\boldsymbol{a}^{(i)} = \mathsf{Expand}(s_a^{(i)})$, $\boldsymbol{b}^{(i)} = \mathsf{Expand}(s_b^{(i)})$, $\boldsymbol{a}^{(i)'} = \mathsf{Expand}(s_a^{(i)'})$, $\boldsymbol{\ell}^{(i)} = \mathsf{Expand}(s_\ell^{(i)})$ and $\boldsymbol{\ell}^{(i)'} = \mathsf{Expand}(s_\ell^{(i)'})$.
  2. Every pair of parties (P_i, P_j) calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending $s_a^{(i)}$ and P_j sending $s_b^{(i)}$, and it sends back $\boldsymbol{u}_{i,j}$ to P_i and $\boldsymbol{v}_{j,i}$ to P_j, such that $\boldsymbol{u}_{i,j} + \boldsymbol{v}_{j,i} = \boldsymbol{a}^{(i)} * \boldsymbol{b}^{(j)}$.
  3. Every pair of parties (P_i, P_j) calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending $s_a^{(i)'}$ and P_j sending $s_b^{(i)}$, and it sends back $\boldsymbol{u}'_{i,j}$ to P_i and $\boldsymbol{v}'_{j,i}$ to P_j, such that $\boldsymbol{u}'_{i,j} + \boldsymbol{v}'_{j,i} = \boldsymbol{a}^{(i)'} * \boldsymbol{b}^{(j)}$.
- To get the k-th triple ($1 \leq k \leq m_T$):
  1. Let $\langle a_k \rangle, \langle b_k \rangle, \langle \ell_k \rangle$ be the k-th shares from $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle$ and $\langle \boldsymbol{\ell} \rangle$. Each P_i computes $c_k^{(i)} = a_k^{(i)} \cdot b_k^{(i)} + \sum_{j\neq i}(\boldsymbol{u}_{i,j}[k] + \boldsymbol{v}_{i,j}[k])$ as his share of $[c]$. Then the parties locally convert $\langle a_k \rangle, \langle b_k \rangle, \langle \ell_k \rangle$ to $[\![a_k]\!], [\![b_k]\!], [\![\ell_k]\!]$.
  2. The parties compute their shares of $[\ell_k + c_k] = [\ell_k] + [c_k]$ and send them to P_1. P_1 reconstructs $\ell_k + c_k$ and sends it to all the parties.
  3. Each P_i locally computes $[\![c_k]\!] = (\ell_k + c_k) - [\![\ell_k]\!]$. Then the parties get the triple $([\![a_k]\!], [\![b_k]\!], [\![c_k]\!])$.
  4. The parties repeat steps 1-3 on $\langle \boldsymbol{a}' \rangle, \langle \boldsymbol{b} \rangle, \langle \boldsymbol{\ell}' \rangle$ in place of $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle, \langle \boldsymbol{\ell} \rangle$ to obtain $([\![a'_k]\!], [\![b_k]\!], [\![c'_k]\!])$.
  5. The parties output $([\![a_k]\!], [\![b_k]\!], [\![c_k]\!])$.

Figure 11: Protocol for generating multiplication triples.

To prepare random bit sharings, we apply the techniques of [DFK+06]. In particular, the parties prepare a pair of SPDZ sharings $[\![r]\!], [\![r^2]\!]$, where r is a random element in $\mathbb{F}_p$. The parties open r^2 and compute $\sqrt{r^2} \in [0, \ldots, (p-1)/2]$. If $\sqrt{r^2} \neq 0$, $\sqrt{r^2}$ is either r or $-r$, with equal probability. Then, $2^{-1}((\sqrt{r^2})^{-1} \cdot r + 1)$ is uniformly random in $\{0, 1\}$. The SPDZ sharings $[\![r]\!], [\![r^2]\!]$ can be prepared using a similar technique as used in preparing triples. The protocol Π_{bit} for generating sharings of random bits is given in Figure 12.

Protocol Π_{bit}

Random Bits:

– *Setup:*

1. Each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ 4 times, with input Extend, and receives the seeds $s_r^{(i)}, s_r^{(i)'}, s_\ell^{(i)}, s_\ell^{(i)'}$. The outputs define vectors of shares $\langle \boldsymbol{r} \rangle, \langle \boldsymbol{r'} \rangle, \langle \boldsymbol{\ell} \rangle, \langle \boldsymbol{\ell'} \rangle$ such that $\boldsymbol{r}^{(i)} = \mathsf{Expand}(s_r^{(i)})$, $\boldsymbol{r}^{(i)'} = \mathsf{Expand}(s_r^{(i)'})$, $\boldsymbol{\ell}^{(i)} = \mathsf{Expand}(s_\ell^{(i)})$ and $\boldsymbol{\ell}^{(i)'} = \mathsf{Expand}(s_\ell^{(i)'})$.
2. Every unordered pair of parties (P_i, P_j) calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending $s_r^{(i)}$ and P_j sending $s_r^{(j)}$, and it sends back $\boldsymbol{u}_{i,j}$ to P_i and $\boldsymbol{u}_{j,i}$ to P_j, such that $\boldsymbol{u}_{i,j} + \boldsymbol{u}_{j,i} = \boldsymbol{r}^{(i)} * \boldsymbol{r}^{(j)}$.
3. Every unordered pair of parties (P_i, P_j) calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending $s_r^{(i)'}$ and P_j sending $s_r^{(j)'}$, and it sends back $\boldsymbol{u}'_{i,j}$ to P_i and $\boldsymbol{u}'_{j,i}$ to P_j, such that $\boldsymbol{u}'_{i,j} + \boldsymbol{u}'_{j,i} = \boldsymbol{r}^{(i)'} * \boldsymbol{r}^{(j)'}$.

– To get the k-th sharing of a random bit ($1 \le k \le m_B$):

1. Let $\langle r_k \rangle, \langle \ell_k \rangle$ be the k-th shares from $\langle \boldsymbol{r} \rangle, \langle \boldsymbol{\ell} \rangle$. Each P_i computes $R_k^{(i)} = (r_k^{(i)})^2 + 2\sum_{j \neq i} \boldsymbol{u}_{i,j}[k]$ as his share of $[R]$. Then the parties locally convert $\langle r_k \rangle, \langle \ell_k \rangle$ to $[\![r]\!], [\![\ell]\!]$.
2. The parties send their shares of $[\ell_k + R_k]$ to P_1. P_1 reconstructs and sends $\ell_k + R_k$ to all the parties.
3. Each P_i locally computes $[\![R_k]\!] = (\ell_k + R_k) - [\![\ell_k]\!]$. Then the parties get the pair $([\![r_k]\!], [\![R_k]\!])$.
4. The parties repeat steps 1-3 on $\langle \boldsymbol{r'} \rangle, \langle \boldsymbol{\ell'} \rangle$ instead of $\langle \boldsymbol{r} \rangle$ and $\langle \boldsymbol{\ell} \rangle$ to obtain $([\![r'_k]\!], [\![R'_k]\!])$.
5. The parties run Π_{Open} on $[\![R_k]\!]$. If $R_k = 0$, the parties output abort and abort the protocol. Otherwise, each party computes $\sqrt{R_k} \in [1, (p-1)/2]$.
6. Each party computes their shares of $[\![\lambda_k]\!]$ by $[\![\lambda_k]\!] = 2^{-1} \cdot ((\sqrt{R_k})^{-1} \cdot [\![r_k]\!] + 1)$.
7. The parties output $[\![\lambda_k]\!]$.

Figure 12: Protocol for preparing random bit sharings.

We need to verify the correctness of g^Δ, $c = a \cdot b$ for each triple, and the correctness of $[\![r^2]\!]$ for preparing bit sharings. For g^Δ, we sacrifice a random SPDZ sharing $[\![r]\!] = ([r], [\Delta], [\Delta \cdot r])$. Then, we can verify g^Δ by verifying that the product of all the parties' shares of $(g^\Delta)^{[r]} \cdot g^{-[\Delta \cdot r]}$ is equal to 1. To verify $c = a \cdot b$ for each triple, the parties sacrifice another triple $[\![a']\!], [\![b']\!], [\![c']\!]$ and use it to compute a SPDZ sharing $[\![\alpha \cdot ab - \alpha \cdot c]\!]$ for a random field element α, and this value is supposed to be opened as 0. Then we only need to check the opened values using $\Pi_{\mathsf{SPDZ\text{-}MAC}}$. The correctness of $[\![r^2]\!]$ can be verified in a similar way. The complete verification protocol Π_{ver} is given in Figure 13.

Protocol Π_{ver}

Verification:

1. The parties sacrifice one SPDZ sharing $[\![r]\!]$ generated by **Random Values**. Let each P_i's share of $[r]$ be $r^{(i)}$, P_i's share of $[\Delta \cdot r]$ be $m^{(i)}$.
2. Each party computes $\sigma^{(i)} = (g^\Delta)^{r^{(i)}} \cdot g^{-m^{(i)}}$ and calls $\mathcal{F}_{\mathsf{Commit}}$ with input $(\mathsf{commit}, P_i, \sigma^{(i)}, \tau_{\sigma^{(i)}})$.
3. The parties open their commitments and check that $\prod_{i=1}^n \sigma^{(i)} = 1$. If not, the parties output abort and abort the protocol.
4. The parties call $\mathcal{F}_{\mathsf{Coin}}$ to get a random value $\alpha \in \mathbb{F}_p$.
5. For each $([\![a_k]\!], [\![b_k]\!], [\![c_k]\!])$ and $([\![a'_k]\!], [\![b_k]\!], [\![c'_k]\!])$ prepared in **Triples**:
   (a) The parties compute and run Π_{Open} on $[\![e_k]\!] = [\![\alpha \cdot a_k - a'_k]\!]$.
   (b) The parties compute $[\![\tau_k]\!] = [\![\alpha \cdot c_k]\!] - e_k \cdot [\![b_k]\!] - [\![c'_k]\!]$
6. For each $([\![r_k]\!], [\![R_k]\!])$ and $([\![r'_k]\!], [\![R'_k]\!])$ prepared in **Random Bits**:
   (a) The parties compute and run Π_{Open} on $[\![d_k]\!] = [\![\alpha \cdot r_k - r'_k]\!]$.
   (b) The parties compute $[\![\tau'_k]\!] = [\![\alpha^2 \cdot R_k]\!] - d_k^2 - 2d_k \cdot [\![r'_k]\!] - [\![R'_k]\!]$.
7. The parties run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check the MACs on all the opened values, i.e., $\{e_k\}_{k \in [1, m_T]}, \{d_k, R_k\}_{k \in [1, m_B]}$.[1]

8. The parties run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check the MACs on $\{\tau_k\}_{k\in[1,m_T]}, \{\tau'_k\}_{k\in[1,m_B]}$ assuming that each $[\![\tau_k]\!]$ and $[\![\tau'_k]\!]$ has been opened to 0.

[1]Here we do not check the MAC on $\ell_k + c_k$ in the generation of triples and $\ell_k + R_k$ in the generation of random bits. The additive errors on them will lead to additive errors on τ_k, τ'_k, which will be detected in the next step.

Figure 13: The verification protocol.

Lemma 1. *The protocol* Π_{prep} *securely realizes* $\mathcal{F}_{\mathsf{prep}}$ *in the* $\{\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{nVOLE}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$*-hybrid model against a malicious adversary corrupting* $n-1$ *parties.*

We give the proof of this lemma in Appendix C.

The communication cost of Π_{prep} is $(12nm_T + 16nm_B)\lambda$ bits. We give a detailed analysis in Appendix D.

6 Main Protocol

In this section, we provide our MPC protocol Π_{main} in the client-server model, where only clients have inputs and outputs. We assume that the clients are $C_1, \ldots, C_n$ and the servers are $S_1, \ldots, S_n$. Π_{main} consists of a garbling phase and a circuit evaluation phase. The clients and servers run the garbling phase and the circuit evaluation phase in order.

Public Parameters. Let W be the number of wires, W_I be the number of input wires, and W_O be the number of output wires. Let G_A be the number of AND gates and G_X be the number of XOR gates. The public key encryption scheme we use is $\mathsf{PKE}_2 = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ from Section 4.2.

Protocol Π_{Garbling}

Garbling Phase

Let λ be the computational security parameter and κ be the statistical security parameter. All the servers agree on the public parameter $pp \leftarrow \mathsf{Setup}(\lambda, \kappa)$ from the public-key encryption scheme $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ constructed in Section 4.

1. **Initialization.** Set $m_T = 4G_A + 2G_X + 2W_I + 2W_O$, $m_B = W$, $m_R = W + 1$. The servers send $(\mathsf{Init}, m_T, m_R, m_B)$ to $\mathcal{F}_{\mathsf{prep}}$ and receive the outputs from $\mathcal{F}_{\mathsf{prep}}$.
2. **Preparing Mask Sharings.** For each wire w, all the servers take one random sharing of a bit generated by **Random Bits** of $\mathcal{F}_{\mathsf{prep}}$ as $[\![\lambda_w]\!]$. λ_w serves as the mask of the wire value of w.
3. **Preparing Separate MAC Keys.** For each input and output wire w:
   (a) The servers take a triple generated by $\mathcal{F}_{\mathsf{prep}}$ as $[\![\Delta_w]\!], [\![\Delta'_w]\!], [\![\Delta_w \cdot \Delta'_w]\!]$.
   (b) The servers take another triple $[\![a_w]\!], [\![b_w]\!], [\![c_w]\!]$ and run Π_{Mult} on $[\![\Delta_w]\!]$ and $[\![\lambda_w]\!]$ to obtain $[\![\Delta_w \cdot \lambda_w]\!]$.
   (c) The servers run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check the MACs on all the opened values, i.e. $(W_I + W_O)$ pairs of d, e opened in $(W_I + W_O)$ executions of Π_{Mult}.
4. **Revealing Input Masks.** For each input wire attached to each client C_i:
   (a) Each server sends his shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ to C_i.
   (b) C_i reconstructs $\lambda_w, \Delta_w, \Delta'_w, \lambda_w \cdot \Delta_w, \Delta_w \cdot \Delta'_w$. If $\lambda_w \cdot \Delta_w$ is not equal to the product of λ_w and Δ_w, or $\Delta_w \cdot \Delta'_w$ is not equal to the product of Δ_w and Δ'_w, abort the protocol.
5. **Sending Input Wire Values.** For each client C_i and each input wire w attached to C_i, C_i sends $v_w \oplus \lambda_w$ to all the servers, where v_w is the input wire value of w.
6. **Preparing Shares of Wire Labels.** For each wire w, the servers take a random sharing $[\![k_{w,0}]\!]$ generated by **Random Values** of $\mathcal{F}_{\mathsf{prep}}$. Let S_i's share of each $[k_{w,0}]$ be $k_{w,0}^{(i)}$.
7. **Sending Public Keys.**

(a) For each wire w, the servers compute their shares $\{g^{k_{w,0}^{(i)}}\}_{i=1}^n$ of $g^{[k_{w,0}]}$ and send them to S_1. S_1 computes and sends $g^{k_{w,0}}$ to all servers.

(b) The servers use a random sharing $[\![r]\!]$ generated by **Random Values** of $\mathcal{F}_{\mathsf{prep}}$. Then all servers compute their shares of $g^{[r]}$ and send them to S_1. S_1 computes and sends g^r to all the servers.

(c) The servers call $\mathcal{F}_{\mathsf{Coin}}$ to get a random seed in $\mathbb{F}_p$ and expand it to get random values $\theta_1, \ldots, \theta_W \in \mathbb{F}_p$, and locally compute $[\![\tau]\!] = \sum_{i=1}^W \theta_i \cdot [\![k_{w_i,0}]\!] + [\![r]\!]$. Then, the servers run Π_{Open} to open τ.

(d) Each server checks whether $g^\tau = \prod_{i=1}^W (g^{k_{w_i,0}})^{\theta_i} \cdot g^r$. If not, abort the protocol.

(e) All servers locally compute $g^{k_{w,1}} = g^{k_{w,0}} \cdot g^\Delta$ for each wire w.

8. **Garbling the Circuit.** For each gate g with input wires a, b and output wire c, let $f_g : \{0,1\}^2 \to \{0,1\}$ be the function computed by the gate.

(a) **Computing Sharings of Output Labels.** All the servers jointly compute SPDZ sharings of

$$\chi_1 = f_g(0 \oplus \lambda_a, 0 \oplus \lambda_b) \oplus \lambda_c, \qquad \chi_2 = f_g(0 \oplus \lambda_a, 1 \oplus \lambda_b) \oplus \lambda_c,$$

$$\chi_3 = f_g(1 \oplus \lambda_a, 0 \oplus \lambda_b) \oplus \lambda_c, \qquad \chi_4 = f_g(1 \oplus \lambda_a, 1 \oplus \lambda_b) \oplus \lambda_c.$$

Note that $\lambda_w^2 = \lambda_w$ for each wire w.

- For AND gates, servers run Π_{Mult} to compute $[\![\lambda_a \cdot \lambda_b]\!], [\![\lambda_c \cdot \lambda_b]\!], [\![\lambda_a \cdot \lambda_c]\!], [\![\lambda_a \cdot \lambda_b \cdot \lambda_c]\!]$. Note that each χ_j can be viewed as a linear combination of $\{1, \lambda_a, \lambda_b, \lambda_c, \lambda_a \cdot \lambda_b, \lambda_a \cdot \lambda_c, \lambda_b \cdot \lambda_c, \lambda_a \cdot \lambda_b \cdot \lambda_c\}$.
- For XOR gates, note that $\chi_1 = \chi_4 = \lambda_a \oplus \lambda_b \oplus \lambda_c$ and $\chi_2 = \chi_3 = 1 \oplus \lambda_a \oplus \lambda_b \oplus \lambda_c$. All servers run Π_{Mult} to compute $[\![\lambda_a \cdot \lambda_b]\!]$ and then locally compute $[\![\lambda_a \oplus \lambda_b]\!] = [\![\lambda_a]\!] + [\![\lambda_b]\!] - 2 \cdot [\![\lambda_a \cdot \lambda_b]\!]$. Similarly, they get $[\![\lambda_a \oplus \lambda_b \oplus \lambda_c]\!]$ using one call of Π_{Mult}.

(b) **Verification of MACs.** The servers run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check the MACs on all the opened values, i.e. τ and $4G_A + 2G_X$ pairs of d, e opened in $4G_A + 2G_X$ executions of Π_{Mult}.

(c) **Encrypting Output Labels.**

i. All servers locally compute $[\![\chi_j]\!]$ and $[x_{c,j}] = [k_{c,0}] + [\chi_j \cdot \Delta]$, $j = 1, 2, 3, 4$.

ii. Each server S_i encrypts his share $x_{c,1}^{(i)}$ (of $[x_{c,1}]$) by $\mathsf{Enc}(pp, g^{k_{a,0}}, g^{k_{b,0}}, x_{c,1}^{(i)})$, $x_{c,2}^{(i)}$ by $\mathsf{Enc}(pp, g^{k_{a,0}}, g^{k_{b,1}}, x_{c,2}^{(i)})$, $x_{c,3}^{(i)}$ by $\mathsf{Enc}(pp, g^{k_{a,1}}, g^{k_{b,0}}, x_{c,3}^{(i)})$, and $x_{c,4}^{(i)}$ by $\mathsf{Enc}(pp, g^{k_{a,1}}, g^{k_{b,1}}, x_{c,4}^{(i)})$. Then, S_i sends the ciphertexts to S_1.

Figure 14: Protocol for the garbling phase.

Protocol Π_{Eval}

Circuit Evaluation Phase

1. **Revealing Input Labels.** For each input wire w, all the servers send their shares of $[k_{w,v_w \oplus \lambda_w}]$ to S_1. S_1 checks whether $k_{w,v_w \oplus \lambda_w}$ is consistent with the corresponding public key. If not, abort the protocol.

2. **Computing the Circuit.** S_1 computes the circuit gate by gate. For each gate with input wires a, b and output wire c, if S_1 knows $k_{a,v_a \oplus \lambda_a}, k_{b,v_b \oplus \lambda_b}$, he can use them to decrypt all the servers' shares of $k_{c,v_c \oplus \lambda_c}$. Then S_1 computes $g^{k_{c,v_c \oplus \lambda_c}}$ and compares it with $g^{k_{c,0}}$ and $g^{k_{c,1}}$ to learn $v_c \oplus \lambda_c$. If $g^{k_{c,v_c \oplus \lambda_c}}$ is not in $\{g^{k_{c,0}}, g^{k_{c,1}}\}$, abort the protocol.

3. **Sending Outputs.** For each client C_i and each output wire w attached to C_i:

(a) S_1 sends $v_w \oplus \lambda_w$ and $k_{w,v_w \oplus \lambda_w}$ to C_i. Then C_i checks whether they match the public key $g^{k_{w,v_w \oplus \lambda_w}}$. If not, abort the protocol.

(b) Each server sends his shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ to C_i.

(c) C_i reconstructs $\lambda_w, \Delta_w, \Delta'_w, \lambda_w \cdot \Delta_w, \Delta_w \cdot \Delta'_w$. If $\lambda_w \cdot \Delta_w$ is not equal to the product of λ_w and Δ_w, or $\Delta_w \cdot \Delta'_w$ is not equal to the product of Δ_w and Δ'_w, abort the protocol.

(d) C_i computes his output v_w from $v_w \oplus \lambda_w$ and λ_w.

Figure 15: Protocol for the circuit evaluation phase.

Theorem 5. *The protocol* Π_{main} *securely realizes* $\mathcal{F}$ *in the* $\{\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$*-hybrid model against a malicious adversary corrupting upto* n *clients and exactly* $n-1$ *servers.*

We give the proof of this theorem in Appendix E. By letting each party play as a client and a server, the client-server model can be reduced to the standard MPC model. Thus, our protocol achieves malicious security against up to $n-1$ corruptions of all the n parties in the standard MPC model.

The communication cost of Π_{main} is $(20W_I + 18W_O + 8W + 72G_A + 56G_X)n\lambda$ bits in the hybrid model. If we use Π_{prep} to realize $\mathcal{F}_{\mathsf{prep}}$, the execution of Π_{prep} requires communication of $(12nm_T + 16nm_B)\lambda = (24W_I + 24W_O + 16W + 48G_A + 24G_X)n\lambda$ bits, resulting in a total communication of $(44W_I + 42W_O + 24W + 120G_A + 80G_X)n\lambda$ bits. Since $W = G_A + G_X + W_I$, the total communication for the complete protocol is $(68W_I + 42W_O + 144G_A + 104G_X)n\lambda$ bits. We give a detailed analysis in Appendix F.

7 Performance Evaluation

We now demonstrate the efficiency of our protocol. In Section 7.1, we provide a comparison of the concrete communication cost of our protocol Π_{main} with other state-of-the-art protocols. We also implement and benchmark the performance of our protocol, the results of which are given in Section 7.2.

7.1 Cost Analysis

Instatiation of [BCO+21] via Le Mans [RS22]. The LPN-based dishonest majority multiparty garbling protocol of [BCO+21], as proposed in their work, incurred a communication cost of $O(n \cdot \lambda)$ bits per gate *per party*. We observe that, by incorporating the preprocessing functionalities of Le Mans [RS22] as we did, and by requiring only one party to evaluate the garbled circuit, the cost of their protocol can be reduced to $O(\lambda)$ bits per gate per party. In order to ensure a fair comparison (unbiased by the underlying SPDZ functionalities used), we first calculate the communication cost per party of this upgraded version of their protocol, as follows. Note that $k, \ell \in \mathsf{poly}(\lambda)$.

- Garbling protocol Π_{Garble}:
  - Steps 1-4 only require authenticated sharings of random values, which can be realized with cost sublinear in n using PCGs.
  - Step 5(a) requires G_A multiplications of bits.
  - Step 5(b) requires $4k \cdot G_A$ multiplications of bits.
  - Step 5(c) requires $8\ell \cdot G_A$ multiplications of bits.
  - Step 5(d) requires W_O openings of bits.
  - Opening the garbled circuit requires $4\ell \cdot G_A$ openings of bits.
- Evaluation protocol Π_{Evaluate}:
  - Step 2 requires $(1+k) \cdot W_I$ openings of bits.
  - All other steps require sublinear (in n) or zero communication.

Using the results of Le Mans [RS22], each multiplication (respectively, opening) can be achieved with a communication cost of $12n$ (respectively, $2n$) field elements. Thus, the upgraded protocol from [BCO+21] requires a total communication cost of $((12 + 48k + 104\ell) \cdot G_A + (2 + 2k) \cdot W_I + 2 \cdot W_O)\, n$ bits.

Remark 2. *The malicious variant of [RS22] is applicable only in large fields, and it cannot be applied directly to the binary field used in [BCO+21]. For simplicity, we omit this distinction in the comparison with [BCO+21].*

Circuit	Ben-Efraim et al. [BCO+21]	This work
AES-128	768.11	65.33
SHA-256	2709.04	239.66

Table 1: Comparison of the communication cost per party (in MB) incurred in the secure computation of the AES-128 and SHA-256 circuits. The computational security parameter is set to $\lambda = 128$ for both protocols. The statistical security parameter for the protocol of [BCO+21] is set to $\kappa = 80$.

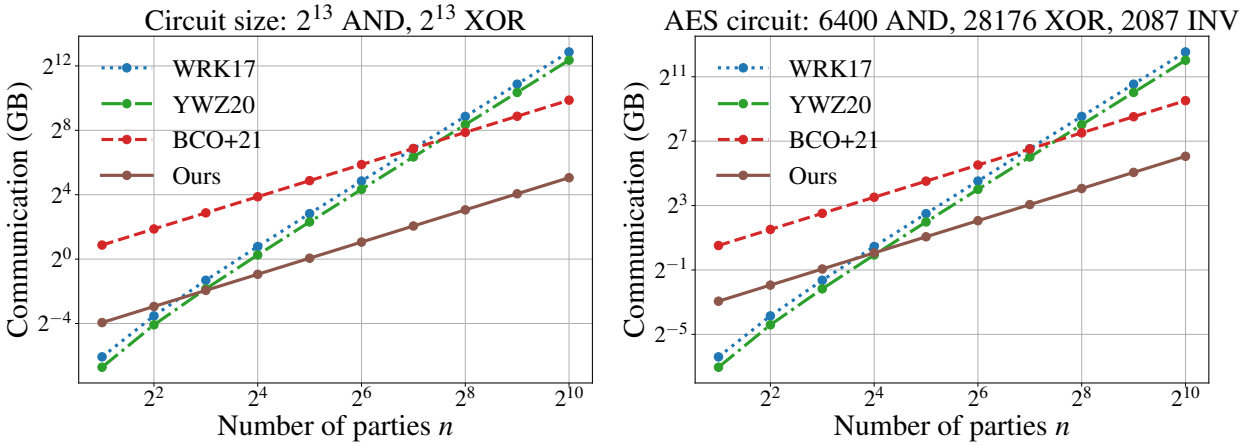

Figure 16: The communication overhead of multiparty garbling protocols. The security parameters are set to $\lambda = 128, \kappa = 80$ in all cases.

Comparison of Concrete Costs. We next calculate and compare the concrete cost of communication per party between our protocol and [BCO+21], including both garbling and evaluation phases, on the AES-128 and SHA-256 circuits. The former has 6400 AND, 28176 XOR, and 2087 INV gates (where each INV(x) can be computed as $1 \oplus x$), while the latter has 22573 AND, 110644 XOR, and 1856 INV gates. The cost of communication per party can be found in Table 1. Considering statistical security parameter $\kappa = 80$ for the protocol of [BCO+21] (where they use s to denote it), we observe that our protocol achieves approximately 11.7× and 11.3× improvements in communication cost on the AES-128 and SHA-256 circuits, respectively.

We also compare with [BGH+23] which achieves $O(|C|)$ communication assuming a strong honest majority. We note that our protocol outperforms that of [BGH+23] in terms of a total cost of communication for up to $n \approx 3500$ parties(we set corruption threshold $t = \lfloor \frac{n-1}{4} \rfloor$ in their protocol for this comparison). This evidence further reinforces the practicality of our protocol.

Finally, we compare with Wang et al. [WRK17] and Yang et al. [YWZ20], which require $O(|C|n^2)$ overall communication. In Figure 16, we compare the end-to-end communication overhead of our protocol with the works of Wang et al. [WRK17], Yang et al. [YWZ20], and Ben-Efraim et al. [BCO+21]. The sizes of the circuits are shown above figures. In these two cases, our protocol always has less communication overhead compared to [BCO+21], and outperforms [WRK17, YWZ20] when there are more than 2^3 to 2^4 parties.

7.2 Implementation and Experiments

We implement and benchmark the performance of our garbled circuits protocol, with a focus on the garbling and evaluation phases, which are our main contributions. We assume a trusted dealer who realizes the functionality $\mathcal{F}_{\mathsf{prep}}$ and distributes the corresponding secret shares to parties in a preprocessing phase. All

experiments are done in an Amazon EC2 c5.24xlarge server with 96 vCPUs and 192 GB RAM. We emulate different network conditions with respect to bandwidth and latency. We assume a latency of 2ms in the LAN setting and 60ms in the WAN setting. For each of these settings, we emulate 10Gbps, 1Gbps, and 100Mbps networks. The implementation is written in C++ and is based on EMP-toolkit [WMK16]. The elliptic curve operations are instantiated by NIST curve P-256 [PUB00] and we use the implementations from the OpenSSL library. In our scheme, the garbling of XOR and AND gates consists of similar operations and takes almost the same amount of running time. Hence, we only record the cost of AND gates in the following experiments.

n	LAN			WAN		
	10Gbps	1Gbps	100Mbps	10Gbps	1Gbps	100Mbps
2	549,1	552,5	597,51	573,15	589,18	619,60
4	588,2	595,15	729,155	645,56	643,54	784,180
8	615,6	639,37	988,362	783,106	776,125	1075,419
16	669,13	726,79	1470,799	1052,227	1016,271	1689,895
32	786,36	925,164	2597,1605	1612,470	1497,555	2964,1850

Table 2: Performance of the garbling protocol in different network conditions. The two numbers in each cell (separated by a comma) represent the time per gate (in 10^{-6} seconds) used for garbling circuits and sending the garbled tables to the evaluator. Each party runs on a single thread.

We first demonstrate the performance of our garbling protocol in different network settings and show the results in Table 2, with the number of parties $n \in \{2, 4, 8, 16, 32\}$. We separately show the average time usage per gate for the garbling phase and the transmission of the garbled circuits, of which time time usage increases with a reduction of bandwidth or an increase in latency. Since our protocol has constant-round communication, the impact of network latency is limited. The garbling time in WAN is only $1.1\times$ to $2\times$ compared to the time in LAN. Our protocol is also communication-efficient. The garbling time in a 100Mbps network compared to that in a 10Gbps network is less than $3.3\times$ in LAN and less than $1.8\times$ in WAN.

Threads	LAN			WAN		
	10Gbps	1Gbps	100Mbps	10Gbps	1Gbps	100Mbps
1	635	667	1206	938	949	1408
2	382	414	934	681	711	1171
4	314	323	831	539	599	1023
8	242	271	776	561	570	976

Table 3: Performance of the garbling phase using different numbers of threads. The numbers are the average time per gate (in 10^{-6} seconds). The number of parties is 12.

Because of the DDH-based encryption scheme, the computational cost of our scheme is higher than schemes that are based purely on symmetric-key operations, though ours has much lower communication complexity. Thus, we explore how multi-threading can improve the speed of garbling. The results are shown in Table 3. In LAN and WAN settings with different bandwidths, we set up 12 parties and increase the number of threads from 1 to 8. In all test cases, the running time of the circuit garbling significantly decreases with the increase of threads, with improvement up to $2.6\times$. Note that the saving brought by the multi-threading implementation is not strictly linear in the number of threads because the garbling phase requires communication and interaction.

Operations	LAN			WAN		
	10Gbps	1Gbps	100Mbps	10Gbps	1Gbps	100Mbps
Generate labels	45	52	196	153	132	257
Compute λs and products	57	73	282	119	125	307
Compute χs	17	17	17	16	17	17
Garble table entries	495	495	492	493	501	492

Table 4: Microbenchmark of the garbling phase. The numbers are the average time per gate (in 10^{-6} seconds). The number of parties is 8 and each party runs on a single thread.

We also show the microbenchmark of the circuit garbling phase in Table 4. The protocol is split into four major components: the generation of labels (k_w, g^{k_w}), the computation of mask bits (λ_a, λ_b) and their products $(\lambda_a\lambda_b, \lambda_a\lambda_c, \lambda_b\lambda_c, \lambda_a\lambda_b\lambda_c)$, the computation of $\{\chi_i\}_{i=1}^4$, and the encryption of table entries. As demonstrated in the table, the main computational bottleneck is the garbling of table entries. Fortunately, this cost can be greatly reduced by multi-threading as shown in the previous Table 3. Note that only the first two operations in Table 4 involve total communication linear in the number of parties and the size of the circuit; hence, their time usage is impacted by network conditions.

n	4	8	16	32
Single-Thread	377	735	1465	2886
Multi-Thread	172	221	249	390
Parallel	94	95	95	188

Table 5: The garbled circuits evaluation time (in 10^{-6} seconds) per gate. The evaluation involves only local operations. In the third row, the number of threads for Evaluator is the same as the number of Garbler. The last row shows the amortized time per gate per circuit when running n MPC instances in parallel and each party acts as Evaluator (using a single thread) in one of the instances. The only exception is when $n = 32$, we only run 16 instances because of the lack of CPU resources.

Finally, we study the performance of the circuit evaluation phase. As shown in the second row of Table 5, the evaluation complexity is linear to the number of parties n. Compared to the garbling phase in Table 2, it becomes a bottleneck when n is large. Hence, we apply multi-threading so that its running time does not grow dramatically while increasing the number of parties (third row). Moreover, notice that the other parties are idle when P_1 is evaluating garbled circuits. We study the amortized cost when parties execute several independent instances of our MPC protocol in parallel, and different parties act as evaluators in these instances. The last row shows that the amortized running time is almost constant when there are n parallel instances.

Acknowledgement. J. Li and Y. Song were supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61361136003.

R. Ostrovsky was supported in part by NSF grants CNS-2246355, CCF-2220450, US-Israel BSF grant 2022370, and by Sunday Group.

References

[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 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 489–518. Springer, 2019.

[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 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 387–416. Springer, 2020.

[BCO+21] Aner Ben-Efraim, Kelong Cong, Eran Omri, Emmanuela Orsini, Nigel P. Smart, and Eduardo Soria-Vazquez. Large scale, actively secure computation from LPN and free-xor garbled circuits. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part III*, volume 12698 of *Lecture Notes in Computer Science*, pages 33–63. Springer, 2021.

[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, *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers*, volume 11959 of *Lecture Notes in Computer Science*, pages 274–302. Springer, 2019.

[BGH+23] Gabrielle Beck, Aarushi Goel, Aditya Hegde, Abhishek Jain, Zhengzhong Jin, and Gabriel Kaptchuk. Scalable multiparty garbling. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2158–2172. ACM, 2023.

[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.

[BLO16] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 578–590. ACM, 2016.

[BLO17] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Efficient scalable constant-round MPC via garbled circuits. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, volume 10625 of *Lecture Notes in Computer Science*, pages 471–498. Springer, 2017.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In Harriet Ortiz, editor, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 503–513. ACM, 1990.

[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, *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings*, volume 11464 of *Lecture Notes in Computer Science*, pages 530–549. Springer, 2019.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptol.*, 13(1):143–202, 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.

[CW79] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2005.

[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, *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[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 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[EGP+23] Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, Yifan Song, and Chenkai Weng. Superpack: Dishonest majority mpc with constant online communication. In *Advances in Cryptology – EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23–27, 2023, Proceedings, Part II*, page 220–250, Berlin, Heidelberg, 2023. Springer-Verlag.

[Elg85] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[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.

[GYKW24] R. Garg, K. Yang, J. Katz, and X. Wang. Scalable mixed-mode mpc. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 109–109, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM J. Comput.*, 28(4):1364–1396, 1999.

[HOSS18a] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, tinykeys for tinyot). In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 86–117. Springer, 2018.

[HOSS18b] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Tinykeys: A new approach to efficient multi-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2018.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 598–628. Springer, 2017.

[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, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 830–842. ACM, 2016.

[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 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189. Springer, 2018.

[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 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 495–512. Springer, 2014.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 319–338. Springer, 2015.

[Mor22] Louis Joel Mordell. On the rational resolutions of the indeterminate equations of the third and fourth degree. In *Proc. Cambridge Phil. Soc.*, volume 21, pages 179–192, 1922.

[NP99] Wim Nevelsteen and Bart Preneel. Software performance of universal hash functions. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the*

Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding, volume 1592 of *Lecture Notes in Computer Science*, pages 24–41. Springer, 1999.

[NZ96] Noam Nisan and David Zuckerman. Randomness is linear in space. *J. Comput. Syst. Sci.*, 52(1):43–52, 1996.

[PS08] Krzysztof Pietrzak and Johan Sjödin. Weak pseudorandom functions in minicrypt. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 423–436. Springer, 2008.

[PUB00] FIPS PUB. Digital signature standard (dss). *Fips pub*, pages 186–192, 2000.

[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 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part I*, volume 13507 of *Lecture Notes in Computer Science*, pages 719–749. Springer, 2022.

[Sti91] Douglas R. Stinson. Universal hashing and authentication codes. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 74–85. Springer, 1991.

[Wei29] André Weil. L'arithmétique sur les courbes algébriques. *Acta mathematica*, 52:281–315, 1929.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 39–56. ACM, 2017.

[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 *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1074–1091. IEEE, 2021.

[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.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.

[YWZ20] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1627–1646. ACM, 2020.

A The Security Model

Real-World Execution. In the real world, there are n parties $P_1, \ldots, P_n$ and an adversary $\mathcal{A}$. We assume that every two parties are connected via a secure synchronous channel so that they can directly send messages to each other. At the beginning of the protocol, $\mathcal{A}$ chooses a set of at most t parties to be corrupted. During the execution of the protocol, the adversary $\mathcal{A}$ controlling corrupted parties interacts with honest parties. At the end of the protocol, the output of the real-world execution includes the inputs and outputs of honest parties and the view of the adversary.

Ideal-World Execution. In the ideal world, the parties interact with a trusted third party $\mathcal{F}$ that receives inputs from the parties, does the computation locally, and sends the outputs to the parties. There is an ideal-world adversary $\mathcal{S}$ who has one-time access to $\mathcal{F}$. $\mathcal{S}$ can send inputs and receive outputs of corrupt clients. $\mathcal{S}$ can also send abort to $\mathcal{F}$ to ask the trusted party to abort. $\mathcal{S}$ emulates the ideal functionality, simulates the honest parties, and interacts with $\mathcal{A}$. The output of the ideal-world execution includes the inputs and outputs of honest clients and servers and the view of the adversary.

We say a protocol Π t-securely realizes $\mathcal{F}$ if there exists an ideal adversary $\mathcal{S}$, such that for all adversary $\mathcal{A}$, the distribution of the output of the real-world execution is computationally indistinguishable from the distribution of the output of the ideal-world execution.

Client-Server Model. In our security proof of the main protocol, we use the *client-server* model. In the client-server model, clients provide inputs to the functionality and receive outputs, and servers can participate in the computation but do not have inputs or outputs in the ideal world. In the real world, the clients only receive input and output wire masks, send the input wire values, and receive the output wire values. Each party may have different roles in the computation. Note that, if every party plays a single client and a single server, this corresponds to a protocol in the standard MPC model. A benefit of the client-server model is that we only need to consider the maximal corruption of servers. At a high level, for an adversary $\mathcal{A}$ which controls $t' < t$ servers, we may construct another adversary $\mathcal{A}'$ which controls additional $t - t'$ servers and behaves as follows:

- For a server corrupted by $\mathcal{A}$, $\mathcal{A}'$ follows the instructions of $\mathcal{A}$. This is achieved by passing messages between this server and other $n - t$ honest servers.
- For a server which is not corrupted by $\mathcal{A}$, but controlled by $\mathcal{A}'$, $\mathcal{A}'$ honestly follows the protocol.

Note that, if a protocol is secure against $\mathcal{A}'$, then this protocol is also secure against $\mathcal{A}$ since the additional $t - t'$ parties controlled by $\mathcal{A}'$ honestly follow the protocol in both cases. Thus, we only need to focus on $\mathcal{A}'$ instead of $\mathcal{A}$, so we can assume that the adversary corrupts exactly $t = n - 1$ servers in the security proof of our main protocol.

B Security Proof for the Encryption Scheme

B.1 Proof of Theorem 3

Proof. We prove this theorem by hybrid arguments. We first prove that for any pair of messages $m_0, m_1 \in \mathbb{F}_p$,

$$\begin{aligned} &\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ =_c\ &\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \end{aligned}$$

holds.

We define the following hybrid distributions:

$$\begin{aligned} H_1 =& \{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ =& \left\{ \begin{matrix} g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_0 \oplus \mathsf{Prg}(\mathsf{Ext}(g^{\mathsf{sk}_1 k_1} \cdot g^{\mathsf{sk}_2 k_2}, s)), \\ s, g^{k_1}, g^{k_2} \end{matrix} : \begin{matrix} \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2 \xleftarrow{\$} \mathbb{F}_p, \\ s \xleftarrow{\$} \{0,1\}^k \end{matrix} \right\} \end{aligned}$$

$$H_2 = \left\{ \begin{matrix} g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_0 \oplus \mathsf{Prg}(\mathsf{Ext}(g^c \cdot g^{\mathsf{sk}_2 k_2}, s)), \\ s, g^{k_1}, g^{k_2} \end{matrix} : \begin{matrix} \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2, c \xleftarrow{\$} \mathbb{F}_p, \\ s \xleftarrow{\$} \{0,1\}^k \end{matrix} \right\}$$

$$H_3 = \left\{ \begin{matrix} g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_0 \oplus \mathsf{Prg}(\mathsf{Ext}(r, s)), \\ s, g^{k_1}, g^{k_2} \end{matrix} : \begin{matrix} \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2 \xleftarrow{\$} \mathbb{F}_p, \\ s \xleftarrow{\$} \{0,1\}^k, r \xleftarrow{\$} G \end{matrix} \right\}$$

$$H_4 = \left\{ \begin{matrix} g^{\mathsf{sk}_1}, \mathsf{sk}_2, m', \\ s, g^{k_1}, g^{k_2} \end{matrix} : \begin{matrix} \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2, c \xleftarrow{\$} \mathbb{F}_p, \\ s \xleftarrow{\$} \{0,1\}^k, m' \xleftarrow{\$} \{0,1\}^k \end{matrix} \right\}$$

$$H_5 = \left\{ \begin{matrix} g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_1 \oplus \mathsf{Prg}(\mathsf{Ext}(r, s)), \\ s, g^{k_1}, g^{k_2} \end{matrix} : \begin{matrix} \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2 \xleftarrow{\$} \mathbb{F}_p, \\ s \xleftarrow{\$} \{0,1\}^k, r \xleftarrow{\$} G \end{matrix} \right\}$$

$$H_6 = \left\{ \begin{matrix} g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_1 \oplus \mathsf{Prg}(\mathsf{Ext}(g^c \cdot g^{\mathsf{sk}_2 k_2}, s)), \\ s, g^{k_1}, g^{k_2} \end{matrix} : \begin{matrix} \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2, c \xleftarrow{\$} \mathbb{F}_p, \\ s \xleftarrow{\$} \{0,1\}^k \end{matrix} \right\}$$

$$H_7 = \left\{ \begin{matrix} g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_1 \oplus \mathsf{Prg}(\mathsf{Ext}(g^{\mathsf{sk}_1 k_1} \cdot g^{\mathsf{sk}_2 k_2}, s)), \\ s, g^{k_1}, g^{k_2} \end{matrix} : \begin{matrix} \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2 \xleftarrow{\$} \mathbb{F}_p, \\ s \xleftarrow{\$} \{0,1\}^k \end{matrix} \right\}$$
$$= \{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}$$

Based on the DDH assumption, we have $H_1 =_c H_2$ and $H_6 =_c H_7$.

Since in H_2, c is uniformly random in F_p, g^c is uniformly random in G and so is $g^c \cdot g^{\mathsf{sk}_2 k_2}$, thus $H_2 =_c H_3$. For the same reason, $H_5 =_c H_6$.

By the property of Ext,

$$\{\mathsf{Ext}(r, s), s : s \xleftarrow{\$} \{0,1\}^k, r \xleftarrow{\$} G\} =_c \{s', s : s' \xleftarrow{\$} \{0,1\}^\lambda, s \xleftarrow{\$} \{0,1\}^k\}.$$

Thus,

$$\begin{aligned} &\{\mathsf{Prg}(\mathsf{Ext}(r, s)), s : s \xleftarrow{\$} \{0,1\}^k, r \xleftarrow{\$} G\} \\ =_c &\{\mathsf{Prg}(s'), s : s' \xleftarrow{\$} \{0,1\}^\lambda, s \xleftarrow{\$} \{0,1\}^k\}. \end{aligned}$$

Since $\mathsf{Prg}(s')$ is uniformly random in $\{0,1\}^k$ when s' is uniformly random in $\{0,1\}^\lambda$, $m_0 \oplus \mathsf{Prg}(\mathsf{Ext}(r, s))$ is also uniformly random in $\{0,1\}^k$. Thus, we have $H_3 =_c H_4$. For the same reason, we have $H_4 =_c H_5$.

Then we can conclude that $H_1 =_c H_7$, which shows that

$$\begin{aligned} &\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ =_c &\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \end{aligned}$$

For the same reason, we also have

$$\begin{aligned} &\{\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\} \\ =_c &\{\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}, \end{aligned}$$

which completes the proof. □

B.2 Proof of Theorem 4

Proof. We prove this theorem by hybrid arguments. We first prove that for any pair of messages $m_0, m_1 \in \mathbb{F}_p$,

$$\begin{aligned}&\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}\\ =_c\ &\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}\end{aligned}$$

holds.

We define the following hybrid distributions:

$$\begin{aligned}H_1 &= \{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}\\ &= \{g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_0 \oplus \mathcal{O}(g^{\mathsf{sk}_1 k_1} \cdot g^{\mathsf{sk}_2 k_2}), g^{k_1}, g^{k_2} : \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2 \xleftarrow{\$} \mathbb{F}_p\}\\ H_2 &= \{g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_0 \oplus \mathcal{O}(g^{c} \cdot g^{\mathsf{sk}_2 k_2}), g^{k_1}, g^{k_2} : \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2, c \xleftarrow{\$} \mathbb{F}_p\}\\ H_3 &= \{g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_0 \oplus \mathcal{O}(r), g^{k_1}, g^{k_2} : \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2 \xleftarrow{\$} \mathbb{F}_p, r \xleftarrow{\$} G\}\\ H_4 &= \{g^{\mathsf{sk}_1}, \mathsf{sk}_2, m', g^{k_1}, g^{k_2} : \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2, c \xleftarrow{\$} \mathbb{F}_p, m' \xleftarrow{\$} \{0,1\}^{2\lambda}\}\\ H_5 &= \{g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_1 \oplus \mathcal{O}(r), g^{k_1}, g^{k_2} : \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2 \xleftarrow{\$} \mathbb{F}_p, r \xleftarrow{\$} G\}\\ H_6 &= \{g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_1 \oplus \mathcal{O}(g^{c} \cdot g^{\mathsf{sk}_2 k_2}), g^{k_1}, g^{k_2} : \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2, c \xleftarrow{\$} \mathbb{F}_p\}\\ H_7 &= \{g^{\mathsf{sk}_1}, \mathsf{sk}_2, m_1 \oplus \mathcal{O}(g^{\mathsf{sk}_1 k_1} \cdot g^{\mathsf{sk}_2 k_2}), g^{k_1}, g^{k_2} : \mathsf{sk}_1, \mathsf{sk}_2, k_1, k_2 \xleftarrow{\$} \mathbb{F}_p\}\\ &= \{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}\end{aligned}$$

Based on the DDH assumption, we have $H_1 =_c H_2$ and $H_6 =_c H_7$.

Since in H_2, c is uniformly random in F_p, g^c is uniformly random in G and so is $g^c \cdot g^{\mathsf{sk}_2 k_2}$, thus $H_2 =_c H_3$. For the same reason, $H_5 =_c H_6$.

Since $\mathcal{O}(r)$ is uniformly random in $\{0,1\}^{2\lambda}$ when r is uniformly random in G, $m_0 \oplus \mathcal{O}(r)$ is also uniformly random in $\{0,1\}^{2\lambda}$. Thus, we have $H_3 =_c H_4$. For the same reason, we have $H_4 =_c H_5$.

Then we can conclude that $H_1 =_c H_7$, which shows that

$$\begin{aligned}&\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}\\ =_c\ &\{\mathsf{pk}_1, \mathsf{sk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}\end{aligned}$$

For the same reason, we also have

$$\begin{aligned}&\{\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_0) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\}\\ =_c\ &\{\mathsf{sk}_1, \mathsf{pk}_2, \mathsf{Enc}(pp, \mathsf{pk}_1, \mathsf{pk}_2, m_1) : (\mathsf{sk}_1, \mathsf{sk}_2, \mathsf{pk}_1, \mathsf{pk}_2) \leftarrow \mathsf{Gen}(pp)\},\end{aligned}$$

which completes the proof. □

C Security Proof for the Preprocessing Protocol

We prove Lemma 1 as follows:

Proof. We prove the security of Π_{prep} by constructing an ideal adversary $\mathcal{S}$. Then we will show that the output in the ideal world is computationally indistinguishable from that in the real world using hybrid arguments.

Without loss of generality, we assume that P_1 is corrupted. We give the construction of the simulator below.

Simulator $\mathcal{S}$

$\mathcal{S}$ simulates Π_{prep} as follows:

Init:

1. $\mathcal{S}$ sends $(\mathsf{Init}, m_T, m_R, m_B)$ to $\mathcal{F}_{\mathsf{prep}}$ and receives g^{Δ}.
2. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nVOLE}}$ to receive Init from each corrupted party.
3. $\mathcal{S}$ sets $\mathsf{DeltaCheck} = \mathsf{MACCheck} = 0$.
4. For each corrupted party P_i, $\mathcal{S}$ receives P_i's share $\Delta^{(i)} \in \mathbb{F}_p$ of $[\Delta]$ from $\mathcal{A}$ and emulates $\mathcal{F}_{\mathsf{nVOLE}}$ to send it to P_i.
5. $\mathcal{S}$ computes the honest party P_j's $g^{\Delta^{(j)}}$ based on corrupted parties' shares of $[\Delta]$ and g^{Δ}.
6. For each corrupted party P_i, $\mathcal{S}$ receives $g^{\Delta^{(i)}}$ from P_i, let the value $\mathcal{S}$ receives be g_i. If the product of the corrupted parties' g_i is not equal to the product of corrupted parties' $g^{\Delta^{(i)}}$, $\mathcal{S}$ sets $\mathsf{DeltaCheck} = 1$. Then $\mathcal{S}$ computes $\tilde{g} = \prod_{P_i \in \mathcal{C}} (g_i / g^{\Delta^{(i)}})$, where $\mathcal{C}$ is the set of corrupted parties.
7. $\mathcal{S}$ samples a random element in $\mathbb{F}_p$ as α.

Random Values:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nVOLE}}$ to receive Extend and $s_r^{(i)}$ from each corrupted party P_i.
2. For each pair of parties (P_i, P_j), if P_i and P_j are both honest, $\mathcal{S}$ does nothing. If P_i is honest but P_j is corrupted, $\mathcal{S}$ receives $\boldsymbol{v}_i^{(j)}$ from $\mathcal{A}$. If P_j is honest but P_i is corrupted, $\mathcal{S}$ receives $\boldsymbol{w}_j^{(i)}$ from $\mathcal{A}$. Finally, if P_i and P_j are both corrupted, $\mathcal{S}$ receives $\boldsymbol{w}_j^{(i)}$ from $\mathcal{A}$ and computes $\boldsymbol{v}_i^{(j)}$ honestly.
3. For each pair of parties (P_i, P_j) where P_i is honest but P_j is corrupted, if $\mathcal{S}$ receives a set I from $\mathcal{A}$, $\mathcal{S}$ samples a random element in $S \backslash I$ as $s_r^{(i)}$. Then $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nVOLE}}$ to send abort and $s_r^{(i)}$ to P_j, sends abort to $\mathcal{F}_{\mathsf{prep}}$, and aborts the protocol on behalf the honest party. After the simulation completes, $\mathcal{S}$ outputs the adversary's view.
4. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nVOLE}}$ to send the output $(s_r^{(i)}, (\boldsymbol{M}_j^{(i)}, \boldsymbol{K}_j^{(i)})_{j \neq i})$ to each corrupted party P_i.
5. $\mathcal{S}$ computes the shares of each $[\![r]\!]$ of corrupted parties.
6. Whenever $(\mathsf{guess}, \boldsymbol{\Delta}')$ is received from $\mathcal{A}$ while $\mathcal{S}$ is emulating $\mathcal{F}_{\mathsf{nVOLE}}$, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{prep}}$, receives Δ from $\mathcal{F}_{\mathsf{prep}}$, and computes the honest party's share of $[\Delta]$ based on Δ and the corrupted parties' shares. If $\boldsymbol{\Delta} = (\Delta^{(1)}, \ldots, \Delta^{(n)}) = \boldsymbol{\Delta}'$, $\mathcal{S}$ aborts the simulation. Otherwise, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nVOLE}}$ to send $(\mathsf{abort}, \boldsymbol{\Delta})$ to $\mathcal{A}$ and follows the protocol to abort the protocol on behalf of the honest party. After the simulation completes, $\mathcal{S}$ outputs the adversary's view.

Triples:

– *Setup:*

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nVOLE}}$ 5 times in the same way as above to generate seeds $s_a^{(i)}, s_b^{(i)}, s_a^{(i)'}, s_\ell^{(i)}, s_\ell^{(i)'}$ for each corrupted party P_i.
2. For the honest party P_i and each corrupted party P_j, $\mathcal{S}$ receives $\tilde{s}_a^{(j)}, \tilde{s}_b^{(j)}$ and $\boldsymbol{u}_{j,i}, \boldsymbol{v}_{j,i}$ from P_j and emulates $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to send $\boldsymbol{u}_{j,i}, \boldsymbol{v}_{j,i}$ back to P_j.
3. For the honest party P_i and each corrupted party P_j, $\mathcal{S}$ receives $\tilde{s}_a^{(j)'}, \tilde{s}_b^{(j)'}$ and $\boldsymbol{u}'_{j,i}, \boldsymbol{v}'_{j,i}$ from P_j and emulates $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to send $\boldsymbol{u}'_{j,i}, \boldsymbol{v}'_{j,i}$ back to P_j.
4. $\mathcal{S}$ sets $\delta_{a_k} = \left(\sum_{j \neq i} \mathsf{Expand}(\tilde{s}_b^{(j)}) - \sum_{j \neq i} \mathsf{Expand}(s_b^{(j)})\right)[k]$,
   $\delta_{b_k} = \left(\sum_{j \neq i} \mathsf{Expand}(\tilde{s}_a^{(j)}) - \sum_{j \neq i} \mathsf{Expand}(s_a^{(j)})\right)[k]$,
   $\delta_{a'_k} = \left(\sum_{j \neq i} \mathsf{Expand}(\tilde{s}_b^{(j)'}) - \sum_{j \neq i} \mathsf{Expand}(s_b^{(j)})\right)[k]$, and
   $\delta_{b'_k} = \left(\sum_{j \neq i} \mathsf{Expand}(\tilde{s}_a^{(j)'}) - \sum_{j \neq i} \mathsf{Expand}(s_a^{(j)'})\right)[k]$.

– To get the k-th triple:

1. $\mathcal{S}$ computes the corrupted parties' shares of $[\ell_k]$ using each corrupted party P_j's $s_\ell^{(j)}$. Then $\mathcal{S}$ computes the corrupted parties' shares of $[c_k]$ by simulating $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ for each pair of corrupted parties (P_i, P_j) assuming that both P_i, P_j correctly send their seeds. Then $\mathcal{S}$ follows the protocol to compute all the corrupted parties' shares of $[\ell_k + c_k]$.
2. For the honest party P_j, $\mathcal{S}$ samples a random element in $\mathbb{F}_p$ as P_j's shares of $[\ell_k + c_k]$. Then $\mathcal{S}$ sends it to P_1 on behalf of P_j.
3. $\mathcal{S}$ computes $\ell_k + c_k$ based on all the parties' shares of $[\ell_k + c_k]$.Then $\mathcal{S}$ receives $\overline{\ell_k + c_k}$ from P_1 and sets $\delta_{\ell_k} = \overline{\ell_k + c_k} - (\ell_k + c_k)$.
4. $\mathcal{S}$ follows the protocol to compute corrupted parties' shares of $[\![c_k]\!] = \overline{\ell_k + c_k} - [\![\ell_k]\!]$.
5. Repeat step 1-3 on ℓ'_k, c'_k instead of ℓ_k, c_k and compute the corresponding $\delta_{\ell'_k}$.
6. If it holds that $\delta_{a_k} = \delta_{b_k} = \delta_{\ell_k} = \delta_{a'_k} = \delta_{b'_k} = \delta_{\ell'_k} = 0$, then $\mathcal{S}$ sets $\tilde{\tau}_k = 0$. Otherwise, $\mathcal{S}$ samples $a_k, b_k, a'_k \in \mathbb{F}_p$ randomly. Let the sum of corrupted parties' shares of $[a_k], [b_k], [a'_k]$ be $\alpha_k, \beta_k, \alpha'_k$. $\mathcal{S}$ sets $\delta_{c_k} = \delta_{\ell_k} - \delta_{a_k} \cdot \alpha_k - \delta_{b_k} \cdot \beta_k$ and $\delta_{c'_k} = \delta_{\ell'_k} - \delta_{a'_k} \cdot \alpha'_k - \delta_{b'_k} \cdot \beta_k$. $\mathcal{S}$ then computes

$$\tilde{\tau}_k = \alpha \cdot (\delta_{a_k} \cdot a_k + \delta_{b_k} \cdot b_k + \delta_{c_k}) - (\delta_{a'_k} \cdot a'_k + \delta_{b'_k} \cdot b_k + \delta_{c'_k}).$$

   Then if $\tilde{\tau}_k = 0$, $\mathcal{S}$ aborts the simulation. Here $\tilde{\tau}_k$ is the value such that $[\tilde{\tau}_k \cdot \Delta]$ is shared among all the parties, and the sharing will be used in the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ while doing verification.

Random Bits:

– *Setup:*

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{nVOLE}}$ 4 times in the same way as above to generate seeds $s_r^{(i)}, s_r^{(i)'}, s_\ell^{(i)}, s_\ell^{(i)'}$ for each party P_i.
2. For the honest party P_i and each corrupted party P_j, $\mathcal{S}$ receives $\tilde{s}_r^{(j)}$ and $\boldsymbol{u}_{j,i}$ from P_j and emulates emulates $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to send $\boldsymbol{u}_{j,i}$ back to P_j.
3. For the honest party P_i and each corrupted party P_j, $\mathcal{S}$ receives $\tilde{s}_r^{(j)'}$ and $\boldsymbol{u}'_{j,i}$ from P_j and emulates emulates $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to send $\boldsymbol{u}'_{j,i}$ back to P_j.
4. $\mathcal{S}$ sets $\delta_{r_k} = 2\left(\sum_{j\neq i} \mathsf{Expand}(\tilde{s}_r^{(j)}) - \sum_{j\neq i} \mathsf{Expand}(s_r^{(j)})\right)[k]$ and $\delta_{r'_k} = 2\left(\sum_{j\neq i} \mathsf{Expand}(\tilde{s}_r^{(j)'}) - \sum_{j\neq i} \mathsf{Expand}(s_r^{(j)'})\right)[k]$.

– To get the k-th sharing for a random bit:

1. $\mathcal{S}$ computes the corrupted parties' shares of $[\ell_k]$ using each corrupted party P_j's $s_\ell^{(j)}$. Then $\mathcal{S}$ computes the corrupted parties' shares of $[R_k]$ by simulating $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ for each pair of corrupted parties (P_i, P_j) assuming that both P_i, P_j correctly send their seeds. Then $\mathcal{S}$ obtains all the corrupted parties' shares of $[\ell_k + R_k]$.
2. For the honest party P_j, $\mathcal{S}$ samples a random element in $\mathbb{F}_p$ as P_j's share of $[\ell_k + R_k]$. Then $\mathcal{S}$ sends it to P_1 on behalf of P_j.
3. $\mathcal{S}$ computes $\ell_k + R_k$ based on all the parties' shares of $[\ell_k + R_k]$. Then $\mathcal{S}$ receives $\overline{\ell_k + R_k}$ from P_1 and sets $\delta_{\ell_k} = \overline{\ell_k + R_k} - (\ell_k + R_k)$.
4. Repeat step 1-3 on ℓ'_k, R'_k instead of ℓ_k, R_k and compute the corresponding $\delta_{\ell'_k}$.
5. $*$ If it holds that $\delta_{r_k} = \delta_{\ell_k} = \delta_{r'_k} = \delta_{\ell'_k} = 0$, then:
   (a) For the honest party P_j, $\mathcal{S}$ samples a random non-zero element in $\mathbb{F}_p$ as r_k and computes $R_k = r_k^2$. Then $\mathcal{S}$ computes P_j's share of $[R_k]$ based on R_k and the corrupted parties' shares of $[R_k]$.
   (b) $\mathcal{S}$ sends P_j's share of $[R_k]$ to P_1 on behalf of P_j.
   (c) $\mathcal{S}$ receives $\tilde{R}_k$ from P_1. If $\tilde{R}_k$ is not equal to R_k, $\mathcal{S}$ sets $\mathsf{MACCheck} = 1$.
   (d) $\mathcal{S}$ sets $\tilde{\tau}'_k = 0$.

   $*$ Otherwise:
   (a) For the honest party P_j, $\mathcal{S}$ samples a random non-zero element in $\mathbb{F}_p$ as r_k and computes $R_k = r_k^2$.

(b) $\mathcal{S}$ samples a random element in $\mathbb{F}_p$ as r'_k. Let the sum of corrupted parties' shares of $[r_k], [r'_k]$ be η_k, η'_k. $\mathcal{S}$ sets $\delta_{R_k} = \delta_{\ell_k} - \delta_{r_k} \cdot \eta_k$ and $\delta_{R'_k} = \delta_{\ell'_k} - \delta_{r'_k} \cdot \eta'_k$. $\mathcal{S}$ then computes

$$\tilde{\tau}'_k = \alpha^2 \cdot (\delta_{r_k} \cdot r_k + \delta_{R_k}) - (\delta_{r'_k} \cdot r'_k + \delta_{R'_k}).$$

Then if $\tilde{\tau}_k = 0$, $\mathcal{S}$ aborts the simulation. Here $\tilde{\tau}'_k$ is the value such that $[\tilde{\tau}'_k \cdot \Delta]$ is shared among all the parties, and the sharing will be used in the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ while doing verification.

(c) $\mathcal{S}$ follows the protocol to compute corrupted parties' shares of $[\![R_k]\!] = \overline{R_k + \ell_k} - [\![\ell_k]\!]$ and obtains each corrupted party P_i's share $R_k^{(i)}$ of $[R_k]$ from the SPDZ sharing. Then $\mathcal{S}$ sets the honest party P_j's share of $[R_k]$ to be $R_k + \delta_{r_k} \cdot r_k + \delta_{R_k} - \sum_{P_i \in C} R_k^{(i)}$ and sends it to P_1 on behalf of P_j.

(d) $\mathcal{S}$ receives $\tilde{R}_k$ from P_1. If $\tilde{R}_k$ is not equal to $R_k + \delta_{r_k} \cdot r_k + \delta_{R_k}$, $\mathcal{S}$ sets $\mathsf{MACCheck} = 1$.

Verification:

1. For each corrupted party P_i, $\mathcal{S}$ follows the protocol to compute P_i's $\sigma^{(i)}$. Then, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, P_i, \sigma^{(i)'}, \tau'_{\sigma^{(i)}})$ from P_i. If the product of $\sigma^{(i)'}$ for all corrupted parties P_i is not equal to the product of $\sigma^{(i)}$ for all corrupted parties P_i, $\mathcal{S}$ sets $\mathsf{DeltaCheck} = 1$.
2. If $\tilde{g} = 1$, $\mathcal{S}$ computes the honest party P_j's $\sigma^{(j)}$ based on the corrupted parties' $\sigma^{(i)}$ and $\prod_{i=1}^n \sigma^{(i)} = 1$. Otherwise, $\mathcal{S}$ samples the honest party P_j's share $r^{(j)}$ of $[r]$ randomly and computes $\sigma^{(j)}$ based on corrupted parties' $\sigma^{(i)}$ and $\prod_{i=1}^n \sigma^{(i)} = \tilde{g}^{r^{(j)}}$. If $\tilde{g} \neq 1$ but $\tilde{g}^{r^{(j)}} = 1$, $\mathcal{S}$ aborts the simulation.
3. For each corrupted party P_i, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(P_i, \tau_{\sigma^{(i)'}})$ to all the parties. For the honest party P_j, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(P_j, \tau_{\sigma^{(j)}})$ to all the parties.
4. For each corrupted party P_i, $\mathcal{S}$ receives $(\mathsf{open}, P_i, \tau_{\sigma^{(i)'}})$ from P_i and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\sigma^{(i)'}, i, \tau_{\sigma^{(i)'}})$ to all the corrupted parties. For the honest party P_j, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\sigma^{(j)}, j, \tau_{\sigma^{(j)}})$ to all the parties.
5. For the honest party P_j, $\mathcal{S}$ follows the protocol to check whether $\sigma^{(j)} \cdot \prod_{i \neq j} \sigma^{(i)'} = 1$. If not, $\mathcal{S}$ follows the protocol to abort the protocol on behalf of P_j and sends abort to $\mathcal{F}_{\mathsf{prep}}$. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.
6. If $\mathsf{DeltaCheck} = 1$, $\mathcal{S}$ aborts the simulation.
7. $\mathcal{S}$ receives $\mathsf{RandCoin}$ from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send α to them. If $\mathcal{S}$ 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, $\mathcal{S}$ outputs the adversary's view.
8. For each $k \in [1, m_T]$:
   (a) $\mathcal{S}$ follows the protocol to compute each corrupted party's share of $[\![e_k]\!]$ and obtains his share of $[e_k]$ from it.
   (b) For the honest party P_j:
   - If $\tilde{\tau}_k = 0$, $\mathcal{S}$ samples a random elements in $\mathbb{F}_p$ as P_j's share of $[e_k]$ and reconstruct e_k based on all the parties' shares of $[e_k]$. Then $\mathcal{S}$ sends P_j's share of $[e_k]$ to P_1 on behalf of P_j.
   - If $\tilde{\tau}_k \neq 0$, $\mathcal{S}$ computes $e_k = \alpha \cdot a_k - a'_k$ and computes P_j's share of $[e_k]$ based on the secret and corrupted parties' shares. Then $\mathcal{S}$ sends P_j's share of $[e_k]$ to P_1 on behalf of P_j.

   (c) $\mathcal{S}$ receives e_k from P_1 and checks whether it is correctly sent. If not, $\mathcal{S}$ sets $\mathsf{MACCheck} = 1$.
9. For each $k \in [1, m_B]$:
   (a) $\mathcal{S}$ follows the protocol to compute each corrupted party's share of $[d_k]$ and obtains his share of $[d_k]$ from it.
   (b) For the honest party P_j:
   - If $\tilde{\tau}'_k = 0$, $\mathcal{S}$ samples a random elements in $\mathbb{F}_p$ as P_j's share of $[d_k]$ and reconstruct d_k based on all the parties' shares of $[d_k]$. Then $\mathcal{S}$ sends P_j's share of $[d_k]$ to P_1 on behalf of P_j.
   - If $\tilde{\tau}'_k \neq 0$, $\mathcal{S}$ computes $d_k = \alpha \cdot r_k - r'_k$ and computes P_j's share of $[d_k]$ based on the secret and corrupted parties' shares. Then $\mathcal{S}$ sends P_j's share of $[d_k]$ to P_1 on behalf of P_j.

(c) $\mathcal{S}$ receives d_k from P_1 and checks whether it is correctly sent. If not, $\mathcal{S}$ sets $\mathsf{MACCheck} = 1$.

10. $\mathcal{S}$ receives $\mathsf{RandCoin}$ from corrupted parties and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send a random seed in $\mathbb{F}_p$ to them. Let the random elements expanded from the seed be $\chi_1, \ldots, \chi_{m_T+2m_B} \in \mathbb{F}_p$. If $\mathcal{S}$ receives abort from $\mathcal{A}$, he sends abort to $\mathcal{F}_{\mathsf{prep}}$ and follows the protocol to abort the protocol on behalf of the honest party. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.
11. $\mathcal{S}$ simulates the first execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as follows:
    (a) $\mathcal{S}$ follows the protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to compute corrupted parties' shares of $[\sigma]$.
    (b) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, P_i, \sigma_i, \tau_{[\sigma]})$ from each corrupted party P_i. $\mathcal{S}$ checks whether the sum of all the corrupted parties' shares of $[\sigma]$ is equal to the sum of σ_i he receives from the corrupted parties. If not, $\mathcal{S}$ sets $\mathsf{MACCheck} = 1$.
    (c)
    - If $\mathsf{MACCheck} = 0$, $\mathcal{S}$ computes the honest party's share of $[\sigma]$ based on the corrupted parties' shares and $\sum_{i=1}^{n}[\sigma] = 0$.
    - If $\mathsf{MACCheck} = 1$, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{prep}}$ and receives Δ from $\mathcal{F}_{\mathsf{prep}}$. For each value A_i opened in $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ protocol:
      * $\mathcal{S}$ 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.
      * $\mathcal{S}$ follows the protocol to compute the honest party's share of $[\sigma]$.

    (d) For each party corrupted P_i, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(P_i, \tau_{[\sigma]})$ to all the corrupted parties. For the honest party P_j, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(P_j, \tau_{[\sigma]})$ to all the corrupted parties.
    (e) For the honest party P_j, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $([\sigma], j, \tau_{[\sigma]})$ to all the corrupted parties. For each corrupted party P_i, $\mathcal{S}$ receives $(\mathsf{open}, P_i, \tau_{[\sigma]})$ from P_i and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\sigma_i, i, \tau_{[\sigma]})$ to all the corrupted parties.
    (f) $\mathcal{S}$ follows the protocol to check whether $\sum_{i=1}^{n}[\sigma] = 0$ holds. If not, $\mathcal{S}$ aborts the protocol on behalf of the honest party. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.
    (g) If $\mathsf{MACCheck} = 1$, $\mathcal{S}$ aborts the simulation.
12. If any of $\tilde{\tau}_k$ or $\tilde{\tau}'_k$ is non-zero, $\mathcal{S}$ sets $\mathsf{MACCheck} = 1$.
13. Repeat steps 11 to simulate the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ assuming that $\{\tau_k\}_{k\in[1,m_T]}, \{\tau'_k\}_{k\in[1,m_B]}$ are opened to be 0. While computing the honest party's share of $[\Delta \cdot \tau_k]$ or $[\Delta \cdot \tau'_k]$ when $\mathsf{MACCheck} = 1$, $\mathcal{S}$ use $\tilde{\tau}_k, \tilde{\tau}'_k$ as τ_k, τ'_k respectively.
14. $\mathcal{S}$ follows the protocol to compute all the corrupted parties' shares of output sharings and sends them to $\mathcal{F}_{\mathsf{prep}}$.
15. $\mathcal{S}$ outputs the adversary's view.

Figure 17: The simulator for Π_{prep}.

We construct the following hybrids:

$\mathbf{Hyb}_0$: In this hybrid, $\mathcal{S}$ runs the protocol honestly. This corresponds to the real-world scenario.

$\mathbf{Hyb}_1$: In this hybrid, $\mathcal{S}$ additionally sets $\mathsf{DeltaCheck} = \mathsf{MACCheck} = 0$ 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, for the honest party P_j, $\mathcal{S}$ doesn't follow the protocol to generate $\Delta^{(j)}$ and compute $g^{\Delta^{(j)}}$ with $\Delta^{(j)}$. Instead, while emulating $\mathcal{F}_{\mathsf{nVOLE}}$ at the beginning of the protocol, $\mathcal{S}$ samples a random element in $\mathbb{F}_p$ as Δ and receives the corrupted parties' shares of $[\Delta]$ from $\mathcal{A}$. Then $\mathcal{S}$ computes $\Delta^{(j)}$ based on Δ and the corrupted parties' share of $[\Delta]$. Besides, $\mathcal{S}$ computes $g^{\Delta^{(j)}}$ with g^{Δ} and corrupted parties' shares of $g^{[\Delta]}$. Note that when Δ is uniformly random in $\mathbb{F}_p$, so is $\Delta^{(j)}$. So we only change the order of generating $\Delta^{(j)}$ and Δ, and the way of generating $g^{\Delta^{(j)}}$ in this hybrid without changing their distributions. These changes make no difference to the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

$\mathbf{Hyb}_3$: In this hybrid, $\mathcal{S}$ additionally sets $\mathsf{DeltaCheck} = 1$ if the product of corrupted parties' $g^{\Delta^{(i)}}$ is not correctly sent. In addition, $\mathcal{S}$ computes $\tilde{g} = \prod_{P_i\in\mathcal{C}}(g_i/g^{\Delta^{(i)}})$, where g_i is the corrupted parties' shares

of $g^{[\Delta]}$ received from them ($\tilde{g}$ is used for simulating the verification process to decide what the honest party P_j's $\sigma^{(j)}$ is). This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same output distribution.

$\mathbf{Hyb}_4$: In this hybrid, for each seed s generated for the honest party by $\mathcal{F}_{\mathsf{nVOLE}}$, $\mathcal{S}$ doesn't generated it while emulating $\mathcal{F}_{\mathsf{nVOLE}}$. Instead, when $\mathcal{S}$ need to compute $\mathsf{Expand}(s)$ to expand it to m field elements on behalf of the honest party or $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, $\mathcal{S}$ samples a uniformly random vector in $\mathbb{F}_p^m$ as the result $\mathsf{Expand}(s)$. Note that each seed generated by $\mathcal{F}_{\mathsf{nVOLE}}$ isn't used anywhere without expanding it, this hybrid is well-defined. 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 $\mathsf{Expand}(s)$ to a random vector for some seed s with a non-negligible probability. Now we construct a PPT algorithm $\mathcal{D}'$ that breaks the security of $\mathsf{Expand}(\cdot)$, which is modeled as a PRG. Concretely, $\mathcal{D}'$ receives a string from the challenger which is either the output of $\mathsf{Expand}(s)$ or a random vector. Then $\mathcal{D}'$ runs $\mathcal{S}$ with the change that $\mathsf{Expand}(s)$ 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 $\mathsf{Expand}(s)$ or sampled randomly. Note that the advantage of $\mathcal{D}'$ is the same as that of $\mathcal{D}$, which breaks the security of the underlying PRG $\mathsf{Expand}(\cdot)$. 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, while $\mathcal{S}$ is emulating $\mathcal{F}_{\mathsf{nVOLE}}$, if $\mathcal{S}$ receives a set I from $\mathcal{A}$, $\mathcal{S}$ aborts the simulation and sends a random seed in $S \backslash I$ to the adversary. This only changes the distribution when $\mathsf{seed}^{(i)}$ happens to be in I. During each simulation of the interaction between corrupted parties and $\mathcal{F}_{\mathsf{nVOLE}}$, $\mathsf{seed}^{(i)}$ isn't sampled before $\mathcal{A}$ sends I to $\mathcal{S}$, the probability that $\mathsf{seed}^{(i)} \in I$ is $|I|/|S|$. Since $\mathcal{A}$ is a PPT adversary, $|I| = \mathsf{poly}(\lambda)$. However $|S|$ is exponential in λ, so the probability is negligible. Thus, the distributions of $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ are statistically indistinguishable.

$\mathbf{Hyb}_6$: In this hybrid, $\mathcal{S}$ samples α in the **Init** process instead of sampling it when $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Coin}}$ while doing verification. We just bring the generation of α earlier, and 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, while generating the triples, $\mathcal{S}$ doesn't generates a random vector as $\mathsf{Expand}(s_a^{(i)})$ for the honest party P_i while emulating $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. Instead, $\mathcal{S}$ generate it when it is needed, i.e. when P_i needs to compute his share of each $[\ell_k + c_k]$ and send it to P_1. Similar for $\mathsf{Expand}(s_b^{(i)})$ and $\mathsf{Expand}(s_a^{(i)'})$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_7$ and $\mathbf{Hyb}_6$ have the same output distribution.

$\mathbf{Hyb}_8$: In this hybrid, while generating the triples, we assume that the honest party is P_i. For each $k \in [1, m_T]$, $\mathcal{S}$ additionally sets

$$\delta_{b_k} = \left(\sum_{j \neq i} \mathsf{Expand}(\tilde{s}_a^{(j)}) - \sum_{j \neq i} \mathsf{Expand}(s_a^{(j)}) \right)[k],$$

and then $\mathcal{S}$ computes

$$\delta_{a_k} = \left(\sum_{j \neq i} \mathsf{Expand}(\tilde{s}_b^{(j)}) - \sum_{j \neq i} \mathsf{Expand}(s_b^{(j)}) \right)[k],$$

$$\delta_{a'_k} = \left(\sum_{j \neq i} \mathsf{Expand}(\tilde{s}_b^{(j)'}) - \sum_{j \neq i} \mathsf{Expand}(s_b^{(j)}) \right)[k], \text{ and}$$

$$\delta_{b'_k} = \left(\sum_{j \neq i} \mathsf{Expand}(\tilde{s}_a^{(j)'}) - \sum_{j \neq i} \mathsf{Expand}(s_a^{(j)'}) \right)[k]$$

in a similar way. Here $s_a^{(j)}$ is the seed generated when $\mathcal{S}$ is emulating $\mathcal{F}_{\mathsf{nVOLE}}$, and $\tilde{s}_a^{(j)}$ is received from P_j when emulating $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ between the honest party P_i and P_j. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_8$ and $\mathbf{Hyb}_7$ have the same output distribution.

$\mathbf{Hyb}_9$: In this hybrid, while generating the triples, $\mathcal{S}$ doesn't compute the honest party P_j's share of each $[\ell_k + c_k]$ by himself. Instead, $\mathcal{S}$ generates a random field element as P_j's share of $[\ell_k + c_k]$ and then computes P_j's share of $[\ell_k]$ by $[\ell_k + c_k] - [c_k]$. Since P_j's share of $[\ell_k]$ is an entry from $\mathsf{Expand}(s_\ell^{(j)})$ which we have replaced the result by a uniformly random vector in $\mathbf{Hyb}_4$, it's a uniformly random element in $\mathbb{F}_p$. Therefore, P_j's share of $[\ell_k + c_k]$ is also a uniformly random variable in $\mathbb{F}_p$, so we only change the order of generating P_j's shares of $[\ell_k + c_k]$ and $[\ell_k]$ without changing their distributions. Thus, $\mathbf{Hyb}_9$ and $\mathbf{Hyb}_8$ have the same output distribution.

$\mathbf{Hyb}_{10}$: In this hybrid, while generating the triples, let $\overline{\ell_k + c_k}$ be what $\mathcal{S}$ received from P_1 for each $k \in [1, m_T]$ after P_1 receives the honest party's share of $[\ell_k + c_k]$, $\mathcal{S}$ additionally sets $\delta_{\ell_k} = \overline{\ell_k + c_k} - (\ell_k + c_k)$. Similarly, $\mathcal{S}$ also computes $\delta_{\ell'_k}$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ have the same output distribution.

$\mathbf{Hyb}_{11}$: In this hybrid, while generating the triples, $\mathcal{S}$ delay the process of computing the honest party P_j's share of each $[c_k]$ and the process of computing each $[\ell_k]$ by $[\ell_k + c_k] - [c_k]$. $\mathcal{S}$ does the above computations after receiving $\overline{\ell_k + c_k}$ from P_1. Since these shares are not used before receiving $\overline{\ell_k + c_k}$ from P_1, these changes don't affect the output distribution. Thus, $\mathbf{Hyb}_{11}$ and $\mathbf{Hyb}_{10}$ have the same output distribution.

$\mathbf{Hyb}_{12}$: In this hybrid, while generating the triples, if it holds that $\delta_{a_k} = \delta_{b_k} = \delta_{\ell_k} = \delta_{a'_k} = \delta_{b'_k} = \delta_{\ell'_k} = 0$, then $\mathcal{S}$ sets $\tilde{\tau}_k = 0$. This means that we can regard that the corrupted parties invoke $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ honestly and the values $\ell_k + c_k, \ell'_k + c'_k$ are opened correctly. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{12}$ and $\mathbf{Hyb}_{11}$ have the same output distribution.

$\mathbf{Hyb}_{13}$: In this hybrid, while generating the triples, if some of $\delta_{a_k}, \delta_{b_k}, \delta_{\ell_k}, \delta_{a'_k}, \delta_{b'_k}, \delta_{\ell'_k}$ is non-zero, $\mathcal{S}$ doesn't obtain the honest party's share of $[a_k], [b_k], [a'_k]$ from the result of expanding the seeds from $\mathcal{F}_{\mathsf{nVOLE}}$. Instead, $\mathcal{S}$ randomly samples $a_k, b_k, a'_k \in \mathbb{F}_p$ and then computes the honest party's share of $[a_k], [b_k], [a'_k]$ with a_k, b_k, a'_k and corrupted parties' shares. Since the honest party's shares of $[a_k], [b_k], [a'_k]$ are all entries from vectors expanded by seeds generated by $\mathcal{S}$ while emulating $\mathcal{F}_{\mathsf{nVOLE}}$ which have been replaced by uniformly random vectors in $\mathbf{Hyb}_4$, a_k, b_k, a'_k are also uniformly random elements in $\mathbb{F}_p$. We only change the order of generating a_k, b_k, a'_k and the honest party's shares of $[a_k], [b_k], [a'_k]$ without changing their distributions. Thus, $\mathbf{Hyb}_{13}$ and $\mathbf{Hyb}_{12}$ have the same output distribution.

$\mathbf{Hyb}_{14}$: In this hybrid, while generating the triples, if some of $\delta_{a_k}, \delta_{b_k}, \delta_{\ell_k}, \delta_{a'_k}, \delta_{b'_k}, \delta_{\ell'_k}$ is non-zero, let the sum of corrupted parties' shares of $[a_k], [b_k], [a'_k]$ be $\alpha_k, \beta_k, \alpha'_k$. $\mathcal{S}$ additionally sets $\delta_{c_k} = \delta_{\ell_k} - \delta_{a_k} \cdot \alpha_k - \delta_{b_k} \cdot \beta_k$ and $\delta_{c'_k} = \delta_{\ell'_k} - \delta_{a'_k} \cdot \alpha'_k - \delta_{b'_k} \cdot \beta_k$. $\mathcal{S}$ additionally computes

$$\tilde{\tau}_k = \alpha \cdot (\delta_{a_k} \cdot a_k + \delta_{b_k} \cdot b_k + \delta_{c_k}) - (\delta_{a'_k} \cdot a'_k + \delta_{b'_k} \cdot b_k + \delta_{c'_k}).$$

Now we explain why $\tilde{\tau}_k$ is shared among all the parties. Note that the additive error of the honest party's share of $[\Delta \cdot c_k]$ is δ_{ℓ_k} plus the error $c_k - a_k \cdot b_k$ on c_k and then multiplied with Δ, where the error $c_k - a_k \cdot b_k$ comes from the error of the honest party's share of $[c_k]$ computed from the output of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. If P_i is honest, P_i's share of $[a_k]$ is $a_k^{(i)} = a_k - \alpha_k$, and his share of $[b_k]$ is $b_k^{(i)} = b_k - \beta_k$. Each invocation of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ between (P_i, P_j) generates additive shares of $a_k^{(i)} \cdot b_k^{(j)}$, where sending incorrect seeds by P_j causes an additive error $\left(\mathsf{Expand}(\tilde{s}_b^{(j)}) - \mathsf{Expand}(s_b^{(j)})\right)[k]$ on $b_k^{(j)}$. So the total error is

$$\sum_{j \neq i} \left(\mathsf{Expand}(\tilde{s}_b^{(j)}) - \mathsf{Expand}(s_b^{(j)})\right)[k] \cdot (a_k - \alpha_k) = \delta_{a_k} \cdot (a_k - \alpha_k)$$

from the invocations of (P_i, P_j). For the same reason, the error generated from invocations of (P_j, P_i) is

$$\sum_{j \neq i} \left(\mathsf{Expand}(\tilde{s}_a^{(j)}) - \mathsf{Expand}(s_a^{(j)})\right)[k] \cdot (b_k - \beta_k) = \delta_{b_k} \cdot (b_k - \beta_k).$$

Thus, the additive error on the honest party's $[c_k]$ is $\delta_{a_k} \cdot (a_k - \alpha_k) + \delta_{b_k} \cdot (b_k - \beta_k)$. Let the corrupted parties' share of $[\![c_k]\!]$ computed by $\overline{c_k + \ell_k} - [\![\ell_k]\!]$, the additive errors on c_k and ℓ_k lead to an error

$$\Delta \cdot (\delta_{a_k} \cdot (a_k - \alpha_k) + \delta_{b_k} \cdot (b_k - \beta_k) + \delta_{\ell_k}) = \Delta \cdot (\delta_{a_k} \cdot a_k + \delta_{b_k} \cdot b_k - \delta_{c_k})$$

on (the secret of) $[\Delta \cdot c_k]$, where $\delta_{c_k} = \delta_{\ell_k} - \delta_{a_k} \cdot \alpha_k - \delta_{b_k} \cdot \beta_k$. The error on $[\Delta \cdot \tau_k]$ is just the error on $\alpha \cdot [\Delta \cdot c_k] - [\Delta \cdot c'_k]$, which is equal to

$$\Delta \cdot \left(\alpha \cdot (\delta_{a_k} \cdot a_k + \delta_{b_k} \cdot b_k + \delta_{c_k}) - (\delta_{a'_k} \cdot a'_k + \delta_{b'_k} \cdot b_k + \delta_{c'_k}) \right).$$

This matches $\Delta \cdot \tilde{\tau}_k$.

The additional computation doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{13}$ have the same output distribution.

$\mathbf{Hyb}_{15}$: In this hybrid, while generating the triples, if some of $\delta_{a_k}, \delta_{b_k}, \delta_{\ell_k}, \delta_{a'_k}, \delta_{b'_k}, \delta_{\ell'_k}$ is non-zero but $\tilde{\tau}_k = 0$, $\mathcal{S}$ aborts the simulation. Recall that $\tilde{\tau}_k = \alpha \cdot (\delta_{a_k} \cdot a_k + \delta_{b_k} \cdot b_k + \delta_{c_k}) - (\delta_{a'_k} \cdot a'_k + \delta_{b'_k} \cdot b_k + \delta_{c'_k})$, where $\delta_{c_k}, \delta_{c'_k}$ are computed by $\delta_{a_k}, \delta_{a'_k}, \delta_{b_k}, \delta_{b'_k}, \alpha_k, \beta_k, \delta_{\ell_k}, \delta_{\ell'_k}$, which are all chosen by corrupted parties. Thus, we can regard that the adversary directly choose $\delta_{a_k}, \delta_{a'_k}, \delta_{b_k}, \delta_{b'_k}, \delta_{c_k}, \delta_{c'_k}$. Since $\delta_{c_k} = \delta_{\ell_k} - \delta_{a_k} \cdot \alpha_k - \delta_{b_k} \cdot \beta_k$ and $\delta_{c'_k} = \delta_{\ell'_k} - \delta_{a'_k} \cdot \alpha'_k - \delta_{b'_k} \cdot \beta_k$, the errors $\delta_{a_k}, \delta_{a'_k}, \delta_{b_k}, \delta_{b'_k}, \delta_{c_k}, \delta_{c'_k}$ can't be all-zero when some of $\delta_{a_k}, \delta_{b_k}, \delta_{\ell_k}, \delta_{a'_k}, \delta_{b'_k}, \delta_{\ell'_k}$ is non-zero.

$\tilde{\tau}_k = 0$ only happens when

$$\delta_{a_k} \cdot a_k + \delta_{b_k} \cdot b_k + \delta_{c_k} = \delta_{a'_k} \cdot a'_k + \delta_{b'_k} \cdot b_k + \delta_{c'_k} = 0$$

or

$$\alpha = \frac{\delta_{a'_k} \cdot a'_k + \delta_{b'_k} \cdot b_k + \delta_{c'_k}}{\delta_{a_k} \cdot a_k + \delta_{b_k} \cdot b_k + \delta_{c_k}}.$$

For the first condition, if $\delta_{a_k}, \delta_{b_k}, \delta_{c_k}$ are not all 0, then we must have either $\delta_{a_k} \neq 0$ or $\delta_{b_k} \neq 0$. Without loss of generality, assume that δ_{a_k} is non-zero. Then a_k must be $-(\delta_{b_k} \cdot b_k + \delta_{c_k})/\delta_{a_k}$. Since a_k, b_k are sampled randomly after the additive errors are fixed, the first condition holds with probability $1/p$, which is negligible. Similarly, if $\delta_{a'_k}, \delta_{b'_k}, \delta_{c'_k}$ are not all 0, then the first condition holds with negligible probability.

In the case that the first condition does not hold, since α is randomly sampled, the second condition holds with negligible probability as well. Thus, the distributions of $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$ are statistically close.

$\mathbf{Hyb}_{16}$: In this hybrid, while generating sharings for random bits, $\mathcal{S}$ doesn't generate a random vector as $\mathsf{Expand}(s_r^{(i)})$ for the honest party P_i while emulating $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. Instead, $\mathcal{S}$ generates it when it is needed, i.e. when P_i needs to compute his share of each $[\ell_k + R_k]$ and send it to P_1. Similar for $\mathsf{Expand}(s_r^{(i)'})$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{16}$ and $\mathbf{Hyb}_{15}$ have the same output distribution.

$\mathbf{Hyb}_{17}$: In this hybrid, while generating sharings for random bits, we assume that the honest party is P_i. For each $k \in [1, m_B]$, $\mathcal{S}$ additionally sets

$$\delta_{r_k} = 2 \left(\sum_{j \neq i} \mathsf{Expand}(\tilde{s}_r^{(j)}) - \sum_{j \neq i} \mathsf{Expand}(s_r^{(j)}) \right) [k].$$

Here each corrupted party P_j's $s_r^{(j)}$ is generated by $\mathcal{S}$ while emulating $\mathcal{F}_{\mathsf{nVOLE}}$, and $\tilde{s}_r^{(j)}$ is received from P_j when emulating $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ between the honest party P_i and P_j. Similar for $s_r^{(j)'}$ and $\tilde{s}_r^{(j)'}$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{17}$ and $\mathbf{Hyb}_{16}$ have the same output distribution.

$\mathbf{Hyb}_{18}$: In this hybrid, while generating sharings for random bits, $\mathcal{S}$ doesn't compute the honest party P_j's share of each $[\ell_k + R_k]$ by himself. Instead, $\mathcal{S}$ generates a random field element as P_j's share of $[\ell_k + R_k]$ and then computes P_j's share of $[\ell_k]$ by $[\ell_k + R_k] - [R_k]$. Since P_j's share of $[\ell_k]$ is an entry from $\mathsf{Expand}(s_\ell^{(j)})$ which we have replaced the result by a uniformly random vector in $\mathbf{Hyb}_4$, it's a uniformly random element in $\mathbb{F}_p$. Therefore, P_j's share of $[\ell_k + R_k]$ is also a uniformly random variable in $\mathbb{F}_p$, so we only change the order of generating P_j's shares of $[\ell_k + R_k]$ and $[\ell_k]$ without changing their distributions. Thus, $\mathbf{Hyb}_{18}$ and $\mathbf{Hyb}_{17}$ have the same output distribution.

$\mathbf{Hyb}_{19}$: In this hybrid, while generating sharings for random bits, let $\overline{\ell_k + R_k}$ be what $\mathcal{S}$ received from P_1 for each $k \in [1, m_B]$ after P_1 receives the honest party's share of $[\ell_k + R_k]$, $\mathcal{S}$ additionally sets $\delta_{\ell_k} = \overline{\ell_k + R_k} - (\ell_k + R_k)$. Similarly, $\mathcal{S}$ also computes $\delta_{\ell'_k}$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{19}$ and $\mathbf{Hyb}_{18}$ have the same output distribution.

$\mathbf{Hyb}_{20}$: In this hybrid, while generating the triples, $\mathcal{S}$ delay the process of computing the honest party P_j's share of each $[R_k]$ and the process of computing each $[\ell_k]$ by $[\ell_k + R_k] - [R_k]$. $\mathcal{S}$ does the above computations after receiving $\overline{\ell_k + R_k}$ from P_1. Since these shares are not used before receiving $\overline{\ell_k + c_k}$ from P_1, these changes don't affect the output distribution. Thus, $\mathbf{Hyb}_{20}$ and $\mathbf{Hyb}_{19}$ have the same output distribution.

$\mathbf{Hyb}_{21}$: In this hybrid, while generating sharings for random bits, if it holds that $\delta_{r_k} = \delta_{r'_k} = \delta_{\ell_k} = \delta_{\ell'_k} = 0$, then $\mathcal{S}$ sets $\tilde{\tau}'_k = 0$. This means that we can regard that the corrupted parties invoke $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ honestly and the values $\ell_k + R_k, \ell'_k + R'_k$ are opened correctly. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{21}$ and $\mathbf{Hyb}_{20}$ have the same output distribution.

$\mathbf{Hyb}_{22}$: In this hybrid, while generating sharings for random bits, if it holds that $\delta_{r_k} = \delta_{r'_k} = \delta_{\ell_k} = \delta_{\ell'_k} = 0$, $\mathcal{S}$ doesn't compute the honest party P_j's share of each $[R_k]$ by himself. Instead, $\mathcal{S}$ randomly samples $r_k \neq 0$, computes $R_k = r_k^2$ and computes P_j's share of $[R_k]$ based on R_k and corrupted parties' shares of $[R_k]$. Then $\mathcal{S}$ sends it to P_1 on behalf of P_j. Correspondingly, the honest party's share of $[r_k]$ is now computed with r_k and the corrupted parties' shares. Since the honest party's share of $[r_k]$ is an entry of a vector expanded from a seed generated by $\mathcal{S}$ while emulating $\mathcal{F}_{\mathsf{nVOLE}}$ which has been replaced by a uniformly random vector in $\mathbf{Hyb}_4$, the honest party's share of $[r_k]$ is a uniformly random element in $\mathbb{F}_p$. Thus, r_k is also a uniformly random element in $\mathbb{F}_p$. If $r_k \neq 0$ in $\mathbf{Hyb}_{21}$, we only change the order of generating the honest party's share of $[r_k]$ and r_k, and the way of generating P_j's share of $[R_k]$. When r_k and corrupted parties' shares of $[R_k]$ are fixed, P_j's share of $[R_k]$ is determined, so the distribution of P_j's shares of $[R_k]$ doesn't change, so output distribution changes if and only if r_k happens to be 0 in $\mathbf{Hyb}_{21}$, but this is only with a negligible probability $1/p$. Thus, the distributions of $\mathbf{Hyb}_{22}$ and $\mathbf{Hyb}_{22}$ are statistically close.

$\mathbf{Hyb}_{23}$: In this hybrid, while generating sharings for random bits, if it holds that $\delta_{r_k} = \delta_{r'_k} = \delta_{\ell_k} = \delta_{\ell'_k} = 0$, $\mathcal{S}$ additionally sets $\mathsf{MACCheck} = 1$ if P_1 doesn't correctly send R_k, i.e. the $\tilde{R}_k$ received from P_1 is not equal to R_k. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{23}$ and $\mathbf{Hyb}_{22}$ have the same output distribution.

$\mathbf{Hyb}_{24}$: In this hybrid, while generating sharings for random bits, if some of $\delta_{r_k}, \delta_{r'_k}, \delta_{\ell_k}, \delta_{\ell'_k}$ is non-zero, $\mathcal{S}$ doesn't obtain the honest party's share of $[r_k], [r'_k]$ from the result of expanding the seeds from $\mathcal{F}_{\mathsf{nVOLE}}$. Instead, $\mathcal{S}$ randomly samples $r_k, r'_k \in \mathbb{F}_p$ and then computes the honest party's share of $[r_k], [r'_k]$ with r_k, r'_k and corrupted parties' shares. Since the honest party's shares of $[r_k], [r'_k]$ are all entries from vectors expanded by seeds generated by $\mathcal{S}$ while emulating $\mathcal{F}_{\mathsf{nVOLE}}$ which have been replaced by uniformly random vectors in $\mathbf{Hyb}_4$, r_k, r'_k are also uniformly random elements in $\mathbb{F}_p$. We only change the order of generating r_k, r'_k and the honest party's shares of $[r_k], [r'_k]$ without changing their distributions. Thus, $\mathbf{Hyb}_{24}$ and $\mathbf{Hyb}_{23}$ have the same output distribution.

$\mathbf{Hyb}_{25}$: In this hybrid, while generating sharings for random bits, if some of $\delta_{r_k}, \delta_{r'_k}, \delta_{\ell_k}, \delta_{\ell'_k}$ is non-zero, let the sum of corrupted parties' shares of $[r_k], [r'_k]$ be η_k, η'_k. $\mathcal{S}$ sets $\delta_{R_k} = \delta_{\ell_k} - \delta_{r_k} \cdot \eta_k$ and $\delta_{R'_k} = \delta_{\ell'_k} - \delta_{r'_k} \cdot \eta'_k$. $\mathcal{S}$ additionally computes

$$\tilde{\tau}'_k = \alpha^2 \cdot (\delta_{r_k} \cdot r_k + \delta_{R_k}) - (\delta_{r'_k} \cdot r'_k + \delta_{R'_k}).$$

Now we explain why $\tilde{\tau}'_k$ is shared among all the parties. Note that the additive error of the honest party's share of $[\Delta \cdot R_k]$ is δ_{ℓ_k} plus the error $R_k - r_k^2$ on R_k and then multiplied with Δ, where the error $R_k - r_k^2$ comes from the error of the honest party's share of $[R_k]$ computed from the output of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. If P_i is honest, P_i's share of $[r_k]$ is $r_k^{(i)} = r_k - \eta_k$. Each invocation of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ between (P_i, P_j) generates additive shares of $2r_k^{(i)} \cdot r_k^{(j)}$, where sending incorrect seeds by P_j causes an additive error $\left(\mathsf{Expand}(\tilde{s}_r^{(j)}) - \mathsf{Expand}(s_r^{(j)})\right)[k]$ on $r_k^{(j)}$. So the total error is

$$2\sum_{j \neq i} \left(\mathsf{Expand}(\tilde{s}_r^{(j)}) - \mathsf{Expand}(s_r^{(j)})\right)[k] \cdot (r_k - \eta_k) = \delta_{r_k} \cdot (r_k - \eta_k)$$

from the invocations of (P_i, P_j). Thus, the additive error on the honest party's $[R_k]$ is $2\delta_{r_k} \cdot (r_k - \eta_k)$. Let the corrupted parties' share of $[\![R_k]\!]$ computed by $\overline{R_k + \ell_k} - [\![\ell_k]\!]$, the additive errors on R_k and ℓ_k lead to an error

$$\Delta \cdot (\delta_{r_k} \cdot (r_k - \eta_k) + \delta_{\ell_k}) = \Delta \cdot (\delta_{r_k} \cdot r_k + \delta_{R_k})$$

on (the secret of) $[\Delta \cdot R_k]$, where $\delta_{R_k} = \delta_{\ell_k} - \delta_{r_k} \cdot \eta_k$. The error on $[\Delta \cdot \tau'_k]$ is just the error on $\alpha^2 \cdot [\Delta \cdot R_k] - [\Delta \cdot R'_k]$, which is equal to

$$\Delta \cdot \left(\alpha \cdot (\delta_{r_k} \cdot r_k + \delta_{R_k}) - (\delta_{r'_k} \cdot r'_k + \delta_{R'_k}) \right).$$

This matches $\Delta \cdot \tilde{\tau}'_k$.

The additional computation doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{25}$ and $\mathbf{Hyb}_{24}$ have the same output distribution.

$\mathbf{Hyb}_{26}$: In this hybrid, while generating sharings for random bits, if some of $\delta_{r_k}, \delta_{r'_k}, \delta_{\ell_k}, \delta_{\ell'_k}$ is non-zero but $\tilde{\tau}'_k = 0$, $\mathcal{S}$ aborts the simulation. Recall that $\tilde{\tau}_k = \alpha \cdot (\delta_{r_k} \cdot r_k + \delta_{R_k}) - (\delta_{r'_k} \cdot r'_k + \delta_{R'_k})$, where $\delta_{R_k}, \delta_{R'_k}$ are computed by $\delta_{r_k}, \delta_{r'_k}, \eta_k, \delta_{\ell_k}, \delta_{\ell'_k}$, which are all chosen by corrupted parties. Thus, we can regard that the adversary directly choose $\delta_{r_k}, \delta_{r'_k}, \delta_{R_k}, \delta_{R'_k}$. Since $\delta_{R_k} = \delta_{\ell_k} - \delta_{r_k} \cdot \eta_k$ and $\delta_{R'_k} = \delta_{\ell'_k} - \delta_{r'_k} \cdot \eta'_k$, the errors $\delta_{r_k}, \delta_{r'_k}, \delta_{c_k}, \delta_{c'_k}$ can't be all-zero when some of $\delta_{r_k}, \delta_{r'_k}, \delta_{\ell_k}, \delta_{\ell'_k}$ is non-zero.

$\tilde{\tau}'_k = 0$ only happens when

$$\delta_{r_k} \cdot r_k + \delta_{R_k} = \delta_{r'_k} \cdot r'_k + \delta_{R'_k} = 0$$

or

$$\alpha = \frac{\delta_{r'_k} \cdot r'_k + \delta_{R'_k}}{\delta_{r_k} \cdot r_k + \delta_{R_k}}.$$

For the first condition, if $\delta_{r_k}, \delta_{R_k}$ are not all 0, then we must have $\delta_{r_k} \neq 0$. Then r_k must be $-\delta_{R_k}/\delta_{r_k}$. Since r_k is sampled randomly after the additive errors are fixed, the first condition holds with probability $1/p$, which is negligible. Similarly, if $\delta_{r'_k}, \delta_{R'_k}$ are not all 0, then the first condition holds with negligible probability.

In the case that the first condition does not hold, since α is randomly sampled, the second condition holds with negligible probability as well. Thus, the distributions of $\mathbf{Hyb}_{26}$ and $\mathbf{Hyb}_{25}$ are statistically close.

$\mathbf{Hyb}_{27}$: In this hybrid, while generating sharings for random bits, if some of $\delta_{r_k}, \delta_{r'_k}, \delta_{\ell_k}, \delta_{\ell'_k}$ is non-zero, $\mathcal{S}$ doesn't follow the computation of the honest party to compute his share of each R_k. Instead, he computes it by $R_k + \delta_{r_k} \cdot r_k + \delta_{R_k} - \sum_{P_i \in \mathcal{C}} R_k^{(i)}$, where $R_k^{(i)}$ is P_i's share of $[R_k]$ obtained from his share of $[\![R_k]\!]$. We have argued in $\mathbf{Hyb}_{25}$ that when all the parties compute $[\![R_k]\!]$ by $\overline{R_k + \ell_k} - [\![\ell_k]\!]$, then they actually get a sharing $[\![R_k + \delta_{r_k} \cdot r_k + \delta_{R_k}]\!]$. Thus, we only change the order of generating the honest party's share of $[R_k + \delta_{r_k} \cdot r_k + \delta_{R_k}]$ and the secret without changing their distributions. Thus, $\mathbf{Hyb}_{27}$ and $\mathbf{Hyb}_{16}$ have the same output distribution.

$\mathbf{Hyb}_{28}$: In this hybrid, while generating sharings for random bits, $\mathcal{S}$ additionally sets $\mathsf{MACCheck} = 1$ if P_1 doesn't correctly open the secret of the sharing $[\![R_k]\!]$ (which is computed as $[\![R_k + \delta_{r_k} \cdot r_k + \delta_{R_k}]\!]$ here). This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{28}$ and $\mathbf{Hyb}_{27}$ have the same output distribution.

$\mathbf{Hyb}_{29}$: In this hybrid, while doing verification, $\mathcal{S}$ additionally sets $\mathsf{DeltaCheck} = 1$ if the product of the value $\sigma^{(i)'}$ committed by each corrupted party P_i is not correct, i.e. $\prod_{P_i \in \mathcal{C}} \sigma^{(i)} \neq \prod_{P_i \in \mathcal{C}} \sigma^{(i)'}$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{29}$ and $\mathbf{Hyb}_{28}$ have the same output distribution.

$\mathbf{Hyb}_{30}$: In this hybrid, while doing verification, if $\tilde{g} = 1$, $\mathcal{S}$ doesn't compute the honest party P_j's $\sigma^{(j)}$ by himself. Instead, he computes $\sigma^{(j)}$ based on each corrupted party P_i's $\sigma^{(i)}$ and $\prod_{i=1}^n \sigma^{(i)} = 1$. Since

$$\prod_{i=1}^n \sigma^{(i)} = (g^\Delta)^{\sum_{i=1}^n r^{(i)}} \cdot g^{\sum_{i=1}^n -m^{(i)}} = g^{\Delta \cdot r} \cdot g^{-\Delta \cdot r} = 1,$$

$\prod_{i=1}^n \sigma^{(i)} = 1$ is guaranteed, and $\tilde{g} = 1$ implies that g^Δ is correctly computed, so we only change the way of generating $\sigma^{(j)}$ without changing its distribution. Thus, $\mathbf{Hyb}_{30}$ and $\mathbf{Hyb}_{29}$ have the same output distribution.

$\mathbf{Hyb}_{31}$: In this hybrid, while doing verification, if $\tilde{g} \neq 1$, $\mathcal{S}$ doesn't get the honest party P_j's share $r^{(j)}$ of $[r]$ from the result of $\mathsf{Expand}(s_r^{(i)})$ and compute P_j's $\sigma^{(j)}$ by himself. Instead, he samples the honest party P_j's share $r^{(j)}$ of $[r]$ randomly and computes $\sigma^{(j)}$ based on corrupted parties' $\sigma^{(i)}$ and $\prod_{i=1}^n \sigma^{(i)} = \tilde{g}^{r^{(j)}}$. Since $\mathcal{S}$ computes g^Δ by computing the product of $g^{\Delta^{(j)}}$ and g_i for each corrupted party P_i on behalf of the honest party P_j, the multiplicative error on $\sigma^{(j)}$ is $(\prod_{i \neq j}(g_i / g^{\Delta^{(i)}}))^{r^{(j)}} = \tilde{g}^{r^{(j)}}$. Thus, the equation $\prod_{i=1}^n \sigma^{(i)} = \tilde{g}^{r^{(j)}}$

holds. P_j's share $r^{(j)}$ of $[r]$ is an entry from the result of Expand with an input seed from $\mathcal{F}_{\mathsf{nVOLE}}$, which has been replaced by a uniformly random vector, so P_j's share $r^{(j)}$ of $[r]$ is a uniformly random field element. Thus, we only change how $r^{(j)}$ and $\sigma^{(j)}$ are generated without changing their distributions. Thus, $\mathbf{Hyb}_{31}$ and $\mathbf{Hyb}_{31}$ have the same output distribution.

$\mathbf{Hyb}_{32}$: In this hybrid, while doing verification, if $\tilde{g} \neq 1$ but $\tilde{g}^{r^{(j)}} = 1$, $\mathcal{S}$ aborts the simulation. Since $r^{(j)}$ is sampled randomly after $\tilde{g}$ is determined, the probability of $\tilde{g}^{r^{(j)}} = 1$ is $1/p$ when $\tilde{g} \neq 1$, which is negligible. Thus, the distributions of $\mathbf{Hyb}_{32}$ and $\mathbf{Hyb}_{31}$ are statistically close.

$\mathbf{Hyb}_{33}$: In this hybrid, while doing verification, after following the protocol to check whether g^{Δ} is correctly sent and decide whether $\mathcal{S}$ needs to aborts the protocol on behalf of the honest party, $\mathcal{S}$ aborts the simulation if $\mathsf{DeltaCheck} = 1$.

Assume that the honest party is P_j. The output only changes when $\mathsf{DeltaCheck} = 1$ but $\sigma^{(j)} \cdot \prod_{P_i \in \mathcal{C}} \sigma^{(i)'} = 1$. Note that $\mathsf{DeltaCheck} = 1$ only happens when $\tilde{g} \neq 1$ or $\prod_{P_i \in \mathcal{C}} \sigma^{(i)} \neq \prod_{P_i \in \mathcal{C}} \sigma^{(i)'}$. If $\tilde{g} \neq 1$ and $\prod_{P_i \in \mathcal{C}} \sigma^{(i)} = \prod_{P_i \in \mathcal{C}} \sigma^{(i)'}$, it must hold that

$$\sigma^{(j)} \cdot \prod_{P_i \in \mathcal{C}} \sigma^{(i)'} = \sigma^{(j)} \cdot \prod_{P_i \in \mathcal{C}} \sigma^{(i)} \neq 1,$$

or $\mathcal{S}$ will abort the simulation as in $\mathbf{Hyb}_{32}$. If $\tilde{g} = 1$ and $\prod_{P_i \in \mathcal{C}} \sigma^{(i)} \neq \prod_{P_i \in \mathcal{C}} \sigma^{(i)'}$, it must hold that

$$\sigma^{(j)} \cdot \prod_{P_i \in \mathcal{C}} \sigma^{(i)'} = 1 \cdot \frac{\prod_{P_i \in \mathcal{C}} \sigma^{(i)'}}{\prod_{P_i \in \mathcal{C}} \sigma^{(i)}} \neq 1.$$

The remaining possible reason why the output distribution changes is that when $\tilde{g} \neq 1$ and $\prod_{P_i \in \mathcal{C}} \sigma^{(i)} \neq \prod_{P_i \in \mathcal{C}} \sigma^{(i)'}$, $\sigma^{(j)} \cdot \prod_{P_i \in \mathcal{C}} \sigma^{(i)'} = 1$ may holds. However,

$$\sigma^{(j)} \cdot \prod_{P_i \in \mathcal{C}} \sigma^{(i)'} = \tilde{g}^{r^{(j)}} \cdot \frac{\prod_{P_i \in \mathcal{C}} \sigma^{(i)'}}{\prod_{P_i \in \mathcal{C}} \sigma^{(i)}}.$$

Thus, there is only one $r^{(j)} \in \mathbb{F}_p$ satisfying $\sigma^{(j)} \cdot \prod_{P_i \in \mathcal{C}} \sigma^{(i)'} = 1$. Since $r^{(j)}$ is sampled randomly after $\tilde{g}$ is determined, the probability that $\sigma^{(j)} \cdot \prod_{P_i \in \mathcal{C}} \sigma^{(i)'} = 1$ is $1/p$, which is negligible.

Thus, the distributions of $\mathbf{Hyb}_{33}$ and $\mathbf{Hyb}_{32}$ are statistically close.

$\mathbf{Hyb}_{34}$: In this hybrid, while doing verification, for each $k \in [1, m_T]$, if $\tilde{\tau}_k = 0$, $\mathcal{S}$ doesn't follow the protocol to compute the honest party P_j's share of $[\![e_k]\!]$ and get his share of $[e_k]$. Instead, $\mathcal{S}$ samples P_j's share of $[e_k]$ randomly in $\mathbb{F}_p$ and computes his share of $[a'_k]$ by $\alpha \cdot [a_k] - [e_k]$. Since P_j's share of $[a'_k]$ is an element from the expansion result of a seed generated by $\mathcal{F}_{\mathsf{nVOLE}}$ which has been replaced by a uniformly random vector in $\mathbf{Hyb}_4$, the share is uniformly random in $\mathbb{F}_p$. Therefore, P_j's share of $[e_k] = \alpha \cdot [a_k] - [a'_k]$ is also uniformly random in $\mathbb{F}_p$. Since P_j's share of $[a'_k]$ is not used in the previous steps of the simulation, we just change the order of generating P_j's shares of $[a'_k]$ and $[e_k]$ without changing their distributions. Thus, $\mathbf{Hyb}_{34}$ and $\mathbf{Hyb}_{33}$ have the same output distribution.

$\mathbf{Hyb}_{35}$: In this hybrid, while doing verification, for each $k \in [1, m_T]$, if $\tilde{\tau}_k \neq 0$, $\mathcal{S}$ doesn't follow the protocol to compute the honest party P_j's share of $[\![e_k]\!]$ and get his share of $[e_k]$. Instead, $\mathcal{S}$ first computes $e_k = \alpha \cdot a_k - a'_k$ and then computes P_j's share of $[e_k]$ based on the secret and the corrupted parties' shares. We only change the order of generating e_k and P_j's share of $[e_k]$ without changing their distributions. Thus, $\mathbf{Hyb}_{35}$ and $\mathbf{Hyb}_{34}$ have the same output distribution.

$\mathbf{Hyb}_{36}$: In this hybrid, while doing verification, for each $k \in [1, m_T]$, $\mathcal{S}$ additionally sets $\mathsf{MACCheck} = 1$ if P_1 doesn't correctly send e_k. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{36}$ and $\mathbf{Hyb}_{35}$ have the same output distribution.

$\mathbf{Hyb}_{37}$: In this hybrid, while doing verification, for each $k \in [1, m_B]$, if $\tilde{\tau}'_k = 0$, $\mathcal{S}$ doesn't follow the protocol to compute the honest party P_j's share of $[\![d_k]\!]$ and get his share of $[d_k]$. Instead, $\mathcal{S}$ samples P_j's share of $[d_k]$ randomly in $\mathbb{F}_p$ and computes his share of $[r'_k]$ by $\alpha \cdot [r_k] - [d_k]$. Since P_j's share of $[r'_k]$ is an

element from the expansion result of a seed generated by $\mathcal{F}_{\mathsf{nVOLE}}$ which has been replaced by a uniformly random vector in $\mathbf{Hyb}_4$, the share is uniformly random in $\mathbb{F}_p$. Therefore, P_j's share of $[d_k] = \alpha \cdot [r_k] - [r'_k]$ is also uniformly random in $\mathbb{F}_p$. Since P_j's share of $[r'_k]$ is not used in the previous steps of the simulation, we just change the order of generating P_j's shares of $[r'_k]$ and $[d_k]$ without changing their distributions. Thus, $\mathbf{Hyb}_{37}$ and $\mathbf{Hyb}_{36}$ have the same output distribution.

$\mathbf{Hyb}_{38}$: In this hybrid, while doing verification, for each $k \in [1, m_B]$, if $\tilde{\tau}_k \neq 0$, $\mathcal{S}$ doesn't follow the protocol to compute the honest party P_j's share of $[\![d_k]\!]$ and get his share of $[d_k]$. Instead, $\mathcal{S}$ first computes $d_k = \alpha \cdot r_k - r'_k$ and then computes P_j's share of $[d_k]$ based on the secret and the corrupted parties' shares. We only change the order of generating d_k and P_j's share of $[d_k]$ without changing their distributions. Thus, $\mathbf{Hyb}_{38}$ and $\mathbf{Hyb}_{37}$ have the same output distribution.

$\mathbf{Hyb}_{39}$: In this hybrid, while doing verification, for each $k \in [1, m_B]$, $\mathcal{S}$ additionally sets $\mathsf{MACCheck} = 1$ if P_1 doesn't correctly send d_k. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{39}$ and $\mathbf{Hyb}_{38}$ have the same output distribution.

$\mathbf{Hyb}_{40}$: In this hybrid, while doing verification, while simulating each execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, $\mathcal{S}$ additionally sets $\mathsf{MACCheck} = 1$ if the sum of the values σ_i committed by each corrupted party P_i is not equal to the sum of their real shares $[\sigma]$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{40}$ and $\mathbf{Hyb}_{39}$ have the same output distribution.

$\mathbf{Hyb}_{41}$: In this hybrid, if $\mathsf{MACCheck} = 0$, while doing verification, $\mathcal{S}$ doesn't compute the honest party's share of $[\sigma]$ by himself. Instead, $\mathcal{S}$ computes the honest party's share of $[\sigma]$ with the corrupted parties' shares and $\sum_{i=1}^{n}[\sigma] = 0$. Since $\mathsf{MACCheck} = 0$, the sum of the shares of each e_k, d_k, R_k is correctly sent for the corresponding opened sharing (where there may be an additive error δ on R_k, but the corresponding SPDZ sharing is $[\![R_k + \delta]\!]$), so $\sum_{i=1}^{n}[\sigma] = 0$ is guaranteed, so we only change the way of generating the honest party's share of $[\sigma]$ without changing its distribution. Thus, $\mathbf{Hyb}_{41}$ and $\mathbf{Hyb}_{40}$ have the same output distribution.

$\mathbf{Hyb}_{42}$: In this hybrid, while simulating the first execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, if $\mathsf{MACCheck} = 1$, $\mathcal{S}$ doesn't directly compute the honest party's share of $[\Delta \cdot A_i]$ for each opened value A_i. Instead, $\mathcal{S}$ first computes each $\Delta \cdot A_i$, and then $\mathcal{S}$ 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, $\mathbf{Hyb}_{42}$ and $\mathbf{Hyb}_{41}$ have the same output distribution.

$\mathbf{Hyb}_{43}$: In this hybrid, while doing verification, before simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, if any of $\tilde{\tau}_k$ or $\tilde{\tau}'_k$ is non-zero, $\mathcal{S}$ additionally sets $\mathsf{MACCheck} = 1$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{43}$ and $\mathbf{Hyb}_{42}$ have the same output distribution.

$\mathbf{Hyb}_{44}$: In this hybrid, while simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, if $\mathsf{MACCheck} = 1$, $\mathcal{S}$ doesn't directly compute the honest party's share of $[\Delta \cdot A_i]$ for each value A_i in $\{\tau_k\}_{k\in[1,m_T]}, \{\tau'_k\}_{k\in[1,m_B]}$ that assumed to be opened to be 0. Instead, $\mathcal{S}$ first computes each $\Delta \cdot A_i$, and then $\mathcal{S}$ computes the honest party's share of $[\Delta \cdot A_i]$ based on the corrupted parties' shares and $\Delta \cdot A_i$, where each $\mathcal{S}$ uses $\{\tilde{\tau}_k\}_{k\in[1,m_T]}, \{\tilde{\tau}'_k\}_{k\in[1,m_B]}$ instead of $\{\tau_k\}_{k\in[1,m_T]}, \{\tau'_k\}_{k\in[1,m_B]}$. As we have argued in $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{25}$, each $[\Delta \cdot \tilde{\tau}_k]$ and $[\Delta \cdot \tilde{\tau}_k]$ is shared among the parties for the verification of MACs in this execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$. Therefore, we just change the way the honest party's share of each $[\Delta \cdot A_i]$ is generated without changing its' distribution. Thus, $\mathbf{Hyb}_{44}$ and $\mathbf{Hyb}_{43}$ have the same output distribution.

$\mathbf{Hyb}_{45}$: In this hybrid, while $\mathcal{S}$ is emulating $\mathcal{F}_{\mathsf{nVOLE}}$, if $\mathcal{S}$ receives a global key query $(\mathsf{guess}, \boldsymbol{\Delta}')$ from $\mathcal{A}$, when $\boldsymbol{\Delta}' = \boldsymbol{\Delta}$, $\mathcal{S}$ aborts the simulation. Otherwise $\mathcal{S}$ sends $\boldsymbol{\Delta}$ to $\mathcal{A}$ and aborts the protocol honestly on behalf of the honest party. This only changes the distribution when $\boldsymbol{\Delta}' = \boldsymbol{\Delta}$, and this only happens when $\mathcal{A}$ correctly guesses Δ. Note that the previous transcripts sent to $\mathcal{A}$ can be generated by $\mathcal{S}$ with g^{Δ} without knowing Δ. If $\mathcal{A}$ correctly guesses Δ, then there exists a PPT algorithm that generates the transcripts in the previous steps and computes Δ from g^{Δ} with a non-negligible probability, which contradicts the DDH assumption. Thus, the distributions of $\mathbf{Hyb}_{45}$ and $\mathbf{Hyb}_{44}$ are computationally indistinguishable.

$\mathbf{Hyb}_{46}$: In this hybrid, while simulating the first execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, after following the protocol to check whether $\sum_{i=1}^{n}[\sigma] = 0$, $\mathcal{S}$ aborts the simulation if $\mathsf{MACCheck} = 1$. The only difference between $\mathbf{Hyb}_{45}$ and $\mathbf{Hyb}_{46}$ 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 assume that the honest party is P_j. 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

$$\begin{aligned}\sigma = &\sum_{i\in[1,n], i\neq j}\left(\sum_{k=1}^{m}\chi_k\cdot[\Delta\cdot A_k] - \Delta^{(i)}\cdot\sum_{k=1}^{m}\chi_k\cdot A_k\right) + \delta_\sigma\\ &+\left(\sum_{k=1}^{m}\chi_k\cdot(\Delta\cdot A_k)^{(j)} - \Delta^{(j)}\cdot\sum_{k=1}^{m}\chi_k\cdot(A_k+\delta_k)\right)\\ =&\delta_\sigma + \Delta^{(j)}\cdot\sum_{k=1}^{m}\chi_k\cdot\delta_k.\end{aligned}$$

If some $\delta_k \neq 0$, since the seed that is expanded to $\chi_1, \ldots, \chi_m$ is sampled after the errors are fixed, and there is only a negligible probability of $1/p$ that $\sum_{k=1}^{m}\chi_k\cdot\delta_k = 0$ if each χ_k is truly random. Thus, if there is a non-negligible probability that $\sum_{k=1}^{m}\chi_k\cdot\delta_k = 0$, then the truly random field elements and the pseudo-random values $\{\chi_k\}_{k=1}^{m}$ can be distinguished by computing $\sum_{k=1}^{m}\chi_k\cdot\delta_k = 0$ with a non-negligible probability, which contradicts the definition of a PRG, so the probability that $\sum_{k=1}^{m}\chi_k\cdot\delta_k = 0$ is negligible. If $\sum_{k=1}^{m}\chi_k\cdot\delta_k \neq 0$, and the certification passes with a non-negligible probability, then the adversary can compute $\Delta^{(j)}$ with the errors and obtains Δ with a non-negligible probability. Since Δ isn't used in the simulation process to compute any transcript sent to any corrupted party before the corrupted parties commit their shares of $[\sigma]$ and all these transcripts can be generated by g^Δ, we can construct a PPT algorithm that runs the previous steps of $\mathcal{S}$ to generates the transcripts sent to $\mathcal{A}$ before the corrupted parties commit their shares of $[\sigma]$ and then runs $\mathcal{A}$ to computes Δ from g^Δ with a non-negligible probability, which contradicts the DDH assumption. Therefore, the distribution only changes with negligible probability.

Thus, the distributions of $\mathbf{Hyb}_{46}$ and $\mathbf{Hyb}_{45}$ are computationally indistinguishable.

$\mathbf{Hyb}_{47}$: In this hybrid, while simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, after following the protocol to check whether $\sum_{i=1}^{n}[\sigma] = 0$, $\mathcal{S}$ aborts the simulation if $\mathsf{MACCheck} = 1$. The only difference between $\mathbf{Hyb}_{47}$ and $\mathbf{Hyb}_{48}$ 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.

As argued in $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{25}$, the SPDZ sharings used for this execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ are $\{[\![\tilde{\tau}_k]\!]\}_{k\in[1,m_T]}$ and $\{[\![\tilde{\tau}'_k]\!]\}_{k\in[1,m_B]}$, where the secret are regarded to be opened as 0. Since $\mathsf{MACCheck} = 1$ only happens when some of $\tilde{\tau}_k$ and $\tilde{\tau}'_k$ are non-zero as in $\mathbf{Hyb}_{43}$, the case is the same as $\mathbf{Hyb}_{46}$ that some secret of a SPDZ sharing is not correctly opened. For the same reason in $\mathbf{Hyb}_{46}$, the distributions of $\mathbf{Hyb}_{47}$ and $\mathbf{Hyb}_{46}$ are computationally indistinguishable.

$\mathbf{Hyb}_{48}$: In this hybrid, when emulating $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, $\mathcal{S}$ delays the generation of the honest party's shares when they are needed. This does not change the output distribution. Thus, the distributions of $\mathbf{Hyb}_{48}$ and $\mathbf{Hyb}_{47}$ are identical.

Note that if the protocol does not abort at the end of the execution and $\mathcal{S}$ 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 $\mathcal{S}$ only need the honest party's shares to prepare the honest party's output of Π_{Prep}.

$\mathbf{Hyb}_{49}$: 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 $\mathcal{S}$ does not abort the simulation. $\mathcal{S}$ no longer generates the honest party's shares when emulating $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. Instead, For each $[\![r]\!]$, $\mathcal{S}$ 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. For each triple $[\![a]\!], [\![b]\!], [\![c]\!]$, $\mathcal{S}$ samples two random values as a, b and computes $c = a \cdot b$. Then $\mathcal{S}$ computes the shares of honest parties using the secrets a, b, c, the MAC key Δ, and the shares of corrupted parties. For each bit sharing $[\![\lambda]\!]$, $\mathcal{S}$ samples a random bit $\lambda \in \{0, 1\}$ and then computes the shares of honest parties using the secret λ, the MAC key Δ, and the shares of corrupted parties. By the correctness of the construction, the distributions of $\mathbf{Hyb}_{49}$ and $\mathbf{Hyb}_{48}$ are identical.

$\mathbf{Hyb}_{50}$: In this hybrid, while emulating $\mathcal{F}_{\mathsf{nVOLE}}$ at the beginning, $\mathcal{S}$ doesn't generate Δ by himself and use it to compute g^Δ. Instead, $\mathcal{S}$ directly gets g^Δ from $\mathcal{F}_{\mathsf{prep}}$. During the simulation, whenever $\mathcal{S}$ needs to

use Δ, he sends abort to $\mathcal{F}_{\mathsf{prep}}$ to obtain Δ. At the end of the simulation, $\mathcal{S}$ sends the corrupted parties' shares of output sharings to $\mathcal{F}_{\mathsf{prep}}$ and let $\mathcal{F}_{\mathsf{prep}}$ compute the honest party's output. Since the process of generating g^{Δ} and the honest party's output by $\mathcal{S}$ himself and by $\mathcal{F}_{\mathsf{prep}}$ is the same, this doesn't change the output distribution. Thus, $\mathbf{Hyb}_{50}$ and $\mathbf{Hyb}_{49}$ have the same output distribution.

Note that $\mathbf{Hyb}_{50}$ is the ideal-world scenario. Thus Π_{Prep} computes $\mathcal{F}_{\mathsf{prep}}$ with computational security.

□

D Cost Analysis for the Preprocessing Phase

We analyze the communication cost of Π_{prep} in the hybrid model. We note that the communication cost of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{nVOLE}}$ is sublinear in its output size when instantiating them from [RS22]. Besides, the number of calls to $\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}$ is independent of the number of random sharings/triples we need to prepare. We will omit the cost of these functionality calls. We will implement the encryption scheme PKE_2 in our MPC construction, so here we assume that p is a 2λ-bit prime and each group element in G can be expressed by a 4λ-bit string.

During the initialization, each party sends a group element $g^{[\Delta]}$ to all the parties, which requires communication of $4n\lambda$ bits.

While preparing SPDZ sharings for random values, the parties compute their shares locally.

While preparing triples, each party sends his shares of $\{[\ell_k + c_k], [\ell'_k + c'_k]\}_{k\in[1,m_T]}$ to P_1 and receives $\{\ell_k + c_k, \ell'_k + c'_k\}_{k\in[1,m_T]}$ from P_1, which requires communication of $8nm_T\lambda$ bits.

While preparing random bit sharings, each party sends his shares of $\{[\ell_k + R_k], [\ell'_k + R'_k], [R_k]\}_{k\in[1,m_B]}$ to P_1 and receives $\{\ell_k + R_k, \ell'_k + R'_k, R_k\}_{k\in[1,m_B]}$ from P_1, which requires communication of $12nm_B\lambda$ bits.

During the verification, each party sends his shares of $\{[e_k]\}_{k\in[1,m_T]}$, $\{[d_k]\}_{k\in[1,m_B]}$ to P_1 and receives $\{e_k\}_{k\in[1,m_T]}, \{d_k\}_{k\in[1,m_B]}$ from P_1, which requires communication of $4nm_T\lambda + 4nm_B\lambda$ bits.

In total, the communication cost of Π_{prep} is about $(12nm_T + 16nm_B)\lambda$ bits.

E Security Proof for the Main Protocol

Proof. We prove the security of Π_{main} by constructing an ideal adversary $\mathcal{S}$. Then we will show that the output in the ideal world is computationally indistinguishable from that in the real world using hybrid arguments. Our simulation is in the client-server model where the adversary corrupts any number of clients and exactly $n-1$ servers.

Without loss of generality, we assume that S_1 is corrupted. We give the construction of the simulator below.

Simulator $\mathcal{S}_{\mathsf{Garbling}}$

Garbling Phase

Then, $\mathcal{S}$ simulates Π_{Garbling} as follows:

1. **Initialization.**
   (a) $\mathcal{S}$ sets $\mathsf{pkCheck} = \mathsf{MACCheck} = 0$ and samples a random element in $\mathbb{F}_p$ as Δ. Then $\mathcal{S}$ computes g^{Δ}.
   (b) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{prep}}$ to receive the shares of $[\Delta]$ of corrupted servers from $\mathcal{A}$ and send g^{Δ} to $\mathcal{A}$.
   (c) $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{prep}}$ to receive $(\mathsf{Init}, m_T, m_R, m_B)$ from each corrupted server, where $m_T = 4G_A + 2G_X + W_I + W_O$, $m_B = W$, $m_R = W + W_I + W_O + 1$.
   (d) $\mathcal{S}$ faithfully emulates $\mathcal{F}_{\mathsf{prep}}$ to receive the shares of the corrupted servers from $\mathcal{A}$ and send the outputs to corrupted servers.
   (e) If abort is received from $\mathcal{A}$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{prep}}$ to send Δ to $\mathcal{A}$ and aborts the protocol honestly on behalf of the honest clients and server. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.
   (f) For the honest server P_j, $\mathcal{S}$ computes $g^{\Delta^{(j)}}$ based on g^{Δ} and corrupted servers' shares of $[\Delta]$.

2. **Preparing Mask Sharings.**
   (a) For each wire w, $\mathcal{S}$ knows the corrupted servers' shares of $[\![\lambda_w]\!]$.
   (b) For each input wire w attached to a corrupted client, $\mathcal{S}$ samples a random bit as λ_w.
3. **Preparing Separate MAC Keys.** For each input and output wire w:
   (a) $\mathcal{S}$ knows the corrupted servers' shares of $[\![\Delta_w]\!], [\![a_w]\!], [\![b_w]\!]$ and follows the protocol to compute their shares of $[\Delta_w - a_w]$ and $[\lambda_w - b_w]$.
   (b) $\mathcal{S}$ samples two random elements in $\mathbb{F}_p$ as the honest server S_j's shares of $[\Delta_w - a_w]$ and $[\lambda_w - b_w]$. Then $\mathcal{S}$ sends them to server S_1 on behalf of S_j.
   (c) $\mathcal{S}$ reconstructs $\Delta_w - a_w, \lambda_w - b_w$ with all the servers' shares. $\mathcal{S}$ receives $\Delta_w - a_w, \lambda_w - b_w$ from server S_1 and checks whether $\Delta_w - a_w$ and $\lambda_w - b_w$ are both correctly reconstructed. If not, $\mathcal{S}$ sets $\mathsf{MACCheck} = 1$.
   (d) $\mathcal{S}$ simulates the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as follows:
      i. $\mathcal{S}$ receives $\mathsf{RandCoin}$ from corrupted servers and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send a random seed in $\mathbb{F}_p$ to them. Let the random elements expanded from the seed be $\chi_1, \ldots, \chi_{2(W_I+W_O)} \in \mathbb{F}_p$. If $\mathcal{S}$ receives abort from $\mathcal{A}$, he aborts the protocol honestly on behalf of the honest clients and server. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.
      ii. $\mathcal{S}$ follows the protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to compute corrupted servers' shares of $[\sigma]$.
      iii. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, S_i, \sigma_i, \tau_{[\sigma]})$ from each corrupted server S_i. $\mathcal{S}$ checks whether the sum of all the corrupted servers' shares of $[\sigma]$ is equal to the sum of σ_i he receives from the corrupted servers. If not, $\mathcal{S}$ sets $\mathsf{MACCheck} = 1$.
      iv. – If $\mathsf{MACCheck} = 0$, $\mathcal{S}$ computes the honest server's share of $[\sigma]$ based on the corrupted servers' shares and $\sum_{i=1}^{n}[\sigma] = 0$.
      – If $\mathsf{MACCheck} = 1$, for each value A_i opened in $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ protocol:
         A. $\mathcal{S}$ computes $\Delta \cdot A_i$ and computes the honest server's share of $[\Delta \cdot A_i]$ based on the corrupted servers' shares and $\Delta \cdot A_i$. Here each A_i is reconstructed based on all the servers' shares, where corrupted servers' shares are computed by following the protocol.
         B. $\mathcal{S}$ follows the protocol to compute the honest server's share of $[\sigma]$.
      v. For each corrupted server S_i, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(S_i, \tau_{[\sigma]})$ to all the corrupted servers. For the honest server S_j, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(S_j, \tau_{[\sigma]})$ to all the corrupted servers.
      vi. For the honest server S_j, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $([\sigma], j, \tau_{[\sigma]})$ to all the corrupted servers. For each corrupted server S_i, $\mathcal{S}$ receives $(\mathsf{open}, S_i, \tau_{[\sigma]})$ from server S_i and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\sigma_i, i, \tau_{[\sigma]})$ to all the corrupted servers.
      vii. $\mathcal{S}$ follows the protocol to check whether $\sum_{i=1}^{n}[\sigma] = 0$ holds. If not, $\mathcal{S}$ aborts the protocol on behalf of the honest server. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.
      viii. If $\mathsf{MACCheck} = 1$, $\mathcal{S}$ aborts the simulation.
4. **Revealing Input Masks.** For each input wire w attached to each client C_i:
   (a) – If client C_i is honest, $\mathcal{S}$ computes the corrupted servers' shares of $[\lambda_w \cdot \Delta_w]$. $\mathcal{S}$ receives the corrupted servers' shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ and checks whether the sum of their shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ are all correct. If not, $\mathcal{S}$ aborts the protocol on behalf of the C_i. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.
   – If client C_i is corrupted:
      i. $\mathcal{S}$ computes the honest server S_j's share of $[\lambda_w]$ based on λ_w and the corrupted servers' shares. Then $\mathcal{S}$ follows the protocol to compute corrupted servers' shares of $[\lambda_w \cdot \Delta_w]$.
      ii. $\mathcal{S}$ samples two random elements in $\mathbb{F}_p$ as the honest server S_j's shares of $[\Delta_w], [\Delta'_w]$. Then $\mathcal{S}$ reconstructs Δ_w, Δ'_w with all the serves' shares, computes $\lambda_w \cdot \Delta_w, \Delta_w \cdot \Delta'_w$, and then computes S_j's shares of $[\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ based on the secrets and the corrupted servers' shares.
      iii. $\mathcal{S}$ sends the honest server S_j's share of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ to client C_i.
5. **Sending Input Wire Values.**
   (a) For each input wire w attached to an honest client, $\mathcal{S}$ samples a random bit as $v_w \oplus \lambda_w$, where v_w is the value of wire w.

(b) For each honest client C_i and each input wire w attached to client C_i, $\mathcal{S}$ sends $v_w \oplus \lambda_w$ to all the corrupted servers on behalf of client C_i. For each corrupted client C_j and each input wire w attached to client C_j, $\mathcal{S}$ receives $v_w \oplus \lambda_w$ from client C_j.

(c) For each corrupted client C_i and each input wire w attached to client C_i, $\mathcal{S}$ computes C_i's input $v_w = (v_w \oplus \lambda_w) \oplus \lambda_w$ of wire w.

6. **Preparing Shares of Wire Labels.** For each wire w, $\mathcal{S}$ knows the corrupted servers' shares of $[\![k_{w,0}]\!]$.

7. **Sending Public Keys.**

(a) $\mathcal{S}$ randomly samples a seed s in $\mathbb{F}_p$ and let the random elements expanded from the seed be $\theta_1, \ldots, \theta_W$. Then $\mathcal{S}$ randomly samples an element in $\mathbb{F}_p$ as $\tau = \sum_{i=1}^{W} \theta_i \cdot k_{w_i,0} + r$.

(b) $\mathcal{S}$ follows the protocol to compute the corrupted servers' shares of $[\tau]$ and use them to compute the honest server S_j's share $\tau^{(j)}$.

(c) For each wire w except input wires, $\mathcal{S}$ samples a random bit as $v_w \oplus \lambda_w$, where v_w is the value of wire w.

(d) For each wire w, $\mathcal{S}$ samples a random element in $\mathbb{F}_p$ as the honest server S_j's share $k^{(j)}_{w,v_w\oplus\lambda_w}$ of $[k_{w,v_w\oplus\lambda_w}]$ and then computes $g^{k^{(j)}_{w,v_w\oplus\lambda_w}}$.

(e) For each wire w, $\mathcal{S}$ computes the honest server S_j's share $g^{k^{(j)}_{w,0}} = g^{k^{(j)}_{w,v_w\oplus\lambda_w}} \cdot (g^{\Delta^{(j)}})^{-(v_w\oplus\lambda_w)}$.

(f) For each wire w, $\mathcal{S}$ sends $g^{k^{(j)}_{w,0}}$ to server S_1 on behalf of the honest server S_j.

(g) For each wire w, $\mathcal{S}$ follows the protocol to compute each corrupted server S_i's shares of $g^{[k_{w,0}]}$. Then $\mathcal{S}$ reconstructs $g^{k_{w,0}}$ based on all the servers' shares. Then $\mathcal{S}$ receives $g^{k_{w,0}}$ from S_1. If it is not correctly sent, $\mathcal{S}$ sets $\mathsf{pkCheck} = 1$.

(h) $\mathcal{S}$ computes the honest server S_j's share $g^{r^{(j)}}$ of $g^{[r]}$ based on $g^{\tau^{(j)}} = \prod_{i=1}^{W} (g^{k^{(j)}_{w_i,0}})^{\theta_i} \cdot g^{r^{(j)}}$. Then $\mathcal{S}$ sends $g^{r^{(j)}}$ to server S_1 on behalf of server S_j.

(i) $\mathcal{S}$ follows the protocol to compute all the corrupted servers' shares of $g^{[r]}$. Then $\mathcal{S}$ computes g^r based on all the servers' shares of $g^{[r]}$.

(j) $\mathcal{S}$ receives g^r from server S_1 and checks whether it is correctly sent. If not, $\mathcal{S}$ sets $\mathsf{pkCheck} = 1$.

(k) $\mathcal{S}$ receives $\mathsf{RandCoin}$ from corrupted servers and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send the random seed $s \in \mathbb{F}_p$ to them. If $\mathcal{S}$ receives abort from $\mathcal{A}$, he aborts the protocol honestly on behalf of the honest server. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.

(l) $\mathcal{S}$ sends the honest server S_j's share of $[\tau]$ to server S_1 on behalf of server S_j.

(m) $\mathcal{S}$ follows the protocol to compute each corrupted server's share of $[\tau]$ and reconstructs τ based on all the servers' shares. Then $\mathcal{S}$ receives τ from server S_1 and checks whether it is correctly sent. If not, $\mathcal{S}$ sets $\mathsf{MACCheck} = 1$.

(n) $\mathcal{S}$ follows the protocol to verify whether $g^\tau = \prod_{i=1}^{W} (g^{k_{w_i,0}})^{\theta_i} \cdot g^r$, where $\{g^{k_{w_i,0}}\}_{i=1}^{W}$ and g^r are received from server S_1. If not, $\mathcal{S}$ aborts the protocol honestly on behalf of the honest server. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.

(o) If $\mathsf{pkCheck} = 1$ and $\mathsf{MACCheck} = 0$, $\mathcal{S}$ aborts the simulation.

(p) $\mathcal{S}$ follows the protocol to compute $g^{k_{w,1}}$ for each wire w.

8. **Garbling the Circuit.** For each gate g with input wires a, b and output wire c, let $f_g : \{0,1\}^2 \to \{0,1\}$ be the function computed by the gate.

(a) **Computing Sharings of Output Labels.** For each execution of Π_{Mult}:

i. $\mathcal{S}$ samples random elements in $\mathbb{F}_p$ as the honest server S_j's shares of $[e], [d]$ and sends them to server S_1 on behalf of server S_j.

ii. $\mathcal{S}$ follows the protocol to compute the corrupted servers' shares of $[e], [d]$ and reconstruct e, d.

iii. $\mathcal{S}$ receives e, d from server S_1 and checks whether they are correctly sent. If not, $\mathcal{S}$ sets $\mathsf{MACCheck} = 1$.

(b) **Verification of MACs.** $\mathcal{S}$ simulates the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as follows:

i. $\mathcal{S}$ receives RandCoin from corrupted servers and emulates $\mathcal{F}_{\mathsf{Coin}}$ to send a random seed in $\mathbb{F}_p$ to them. Let the random elements expanded from the seed be $\chi_1, \ldots, \chi_{2W_O+8G_A+4G_X+1} \in \mathbb{F}_p$. If $\mathcal{S}$ receives abort from $\mathcal{A}$, he aborts the protocol on behalf of the honest server. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.

ii. $\mathcal{S}$ follows the protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to compute corrupted servers' shares of $[\sigma]$.

iii. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, S_i, \sigma_i, \tau_{[\sigma]})$ from each corrupted server S_i. $\mathcal{S}$ checks whether the sum of all the corrupted servers' shares of $[\sigma]$ is equal to the sum of σ_i he receives from the corrupted servers. If not, $\mathcal{S}$ sets $\mathsf{MACCheck} = 1$.

iv. – If $\mathsf{MACCheck} = 0$, $\mathcal{S}$ computes the honest server's share of $[\sigma]$ based on the corrupted servers' shares and $\sum_{i=1}^{n}[\sigma] = 0$.
– If $\mathsf{MACCheck} = 1$, for each value A_i opened in $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ protocol:
A. $\mathcal{S}$ computes $\Delta \cdot A_i$ and computes the honest server's share of $[\Delta \cdot A_i]$ based on the corrupted servers' shares and $\Delta \cdot A_i$. Among the opened values, τ has been sampled by $\mathcal{S}$, and each other A_i is reconstructed based on all the servers' shares, where the corrupted servers' shares are computed by following the protocol.
B. $\mathcal{S}$ follows the protocol to compute the honest server's share of $[\sigma]$.

v. For each corrupted server S_i, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(S_i, \tau_{[\sigma]})$ to all the corrupted servers. For the honest server S_j, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(S_j, \tau_{[\sigma]})$ to all the corrupted servers.

vi. For the honest server S_j, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $([\sigma], j, \tau_{[\sigma]})$ to all the corrupted servers. For each corrupted server S_i, $\mathcal{S}$ receives $(\mathsf{open}, S_i, \tau_{[\sigma]})$ from server S_i and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\sigma_i, i, \tau_{[\sigma]})$ to all the corrupted servers.

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

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

(c) **Encrypting Output Labels.**

i. $\mathcal{S}$ follows the protocol to compute corrupted servers' shares of $[x_{c,v_c\oplus\lambda_c}]$. Then, $\mathcal{S}$ sets $x_{c,v_c\oplus\lambda_c} = k_{c,v_c\oplus\lambda_c}$ and computes the honest server S_j's share $x^{(j)}_{c,v_c\oplus\lambda_c}$ of $[x_{c,v_c\oplus\lambda_c}]$ based on the secret and corrupted servers' shares.

ii. For the honest server S_j, $\mathcal{S}$ computes $\mathsf{Enc}(pp, g^{k_{a,v_a\oplus\lambda_a}}, g^{k_{b,v_b\oplus\lambda_b}}, x^{(j)}_{c,v_c\oplus\lambda_c})$. Then, $\mathcal{S}$ takes 3 random elements in $\{0,1\}^{2\lambda} \times G^2$ as the other 3 cipher-texts encrypted by S_j of this gate. $\mathcal{S}$ then sends the four cipher-texts to server S_1 on behalf of the honest server S_j.

Figure 18: The simulator for the garbling phase.

Simulator $\mathcal{S}_{\mathsf{Eval}}$

Circuit Evaluation Phase

1. For each input wire w, $\mathcal{S}$ sends the honest server S_j's share of $[k_{w,v_w\oplus\lambda_w}]$ to server S_1 on behalf of server S_j.
2. For each output wire w attached to an honest client C_i:
   (a) $\mathcal{S}$ receives $v_w \oplus \lambda_w$ and $k_{w,v_w\oplus\lambda_w}$ from server S_1. $\mathcal{S}$ follows the protocol to check whether $g^{k_{w,v_w\oplus\lambda_w}}$ matches the public key. If not, $\mathcal{S}$ aborts the protocol honestly on behalf of the honest clients and server. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.
   (b) $\mathcal{S}$ sends the corrupted clients' inputs to $\mathcal{F}$ and receives the output wire values v_w for each output wire w attached to a corrupted client. Then, $\mathcal{S}$ computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$ for these wires.
   (c) – If client C_i is honest, $\mathcal{S}$ computes the corrupted servers' shares of $[\lambda_w \cdot \Delta_w]$. $\mathcal{S}$ receives the corrupted servers' shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ and checks whether the sum of their shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ are all correct. If not, $\mathcal{S}$ aborts the protocol on behalf of the C_i. After completing the simulation, $\mathcal{S}$ outputs the adversary's view.
   – If client C_i is corrupted:
   i. $\mathcal{S}$ computes the honest server S_j's share of $[\lambda_w]$ based on λ_w and the corrupted servers' shares. Then $\mathcal{S}$ follows the protocol to compute corrupted servers' shares of $[\lambda_w \cdot \Delta_w]$.

ii. $\mathcal{S}$ samples two random elements in $\mathbb{F}_p$ as the honest server S_j's shares of $[\Delta_w], [\Delta'_w]$. Then $\mathcal{S}$ computes $\lambda_w \cdot \Delta_w, \Delta_w \cdot \Delta'_w$ and then computes S_j's shares of $[\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ based on the secrets and the corrupted servers' shares.

iii. $\mathcal{S}$ sends the honest server S_j's share of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ to client C_i.

3. $\mathcal{S}$ outputs the adversary's view.

Figure 19: The simulator for the circuit evaluation phase.

We construct the following hybrids:

$\mathbf{Hyb}_0$: In this hybrid, $\mathcal{S}$ runs the protocol honestly. This corresponds to the real-world scenario.

$\mathbf{Hyb}_1$: In this hybrid, $\mathcal{S}$ additionally sets $\mathsf{pkCheck} = \mathsf{MACCheck} = 0$. In addition, when emulating $\mathcal{F}_{\mathsf{prep}}$, $\mathcal{S}$ delays the generation of honest server's shares until needed. 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, for each honest client C_i and each input or output wire w attached to client C_i, $\mathcal{S}$ doesn't follow the protocol to compute the honest server S_j's shares of $[\Delta_w - a_w]$ and $[\lambda_w - b_w]$. Instead, $\mathcal{S}$ samples two random elements in $\mathbb{F}_p$ as the honest server S_j's shares of $[\Delta_w - a_w]$ and $[\lambda_w - b_w]$, and then $\mathcal{S}$ computes S_j's shares of $[a_w]$ by $[\Delta_w] - [\Delta_w - a_w]$ and $[b_w]$ by $[\lambda_w] - [\lambda_w - b_w]$. Then $\mathcal{S}$ computes S_j's shares of $[\Delta \cdot a_w], [\Delta \cdot b_w], [\![c_w]\!]$ by reconstructing the a_w, b_w, computing $\Delta \cdot a_w, \Delta \cdot b_w, c_w, \Delta \cdot c_w$, and then computing S_j's shares based on corrupted parties' shares. Since server S_j's shares of $[a_w]$ and $[b_w]$ are uniformly random, so are server S_j's shares of $[\Delta_w - a_w]$ and $[\lambda_w - b_w]$. Hence, we just change the order of generating server S_j's shares of $[\Delta_w - a_w], [\lambda_w - b_w]$ and $[a_w], [b_w]$ without changing their distributions. Since the distributions of S_j's shares of $[\Delta \cdot a_w], [\Delta \cdot b_w], [\![c_w]\!]$ is determined by the distributions of S_j's shares of $[a_w], [b_w]$. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

$\mathbf{Hyb}_3$: In this hybrid, for each input or output wire w, if either of $\Delta_w - a_w, \lambda_w - b_w$ is not sent correctly, $\mathcal{S}$ additional sets $\mathsf{MACCheck} = 1$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same output distribution.

$\mathbf{Hyb}_4$: In this hybrid, for the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ while preparing separate MAC keys, if $\mathsf{MACCheck} = 0$, $\mathcal{S}$ doesn't compute the honest server's share of $[\sigma]$ by himself. Instead, $\mathcal{S}$ computes the honest server's share of $[\sigma]$ with the corrupted servers' shares and $\sum_{i=1}^{n}[\sigma] = 0$. Since $\mathsf{MACCheck} = 0$, each pair of $\Delta_w - a_w, \lambda_w - b_w$ is opened correctly, which guarantees $\sum_{i=1}^{n}[\sigma] = 0$, so we only change the way of generating the honest server's share of $[\sigma]$ without changing its distribution. Thus, $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ have the same output distribution.

$\mathbf{Hyb}_5$: In this hybrid, for the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ while preparing separate MAC keys, if $\mathsf{MACCheck} = 1$, $\mathcal{S}$ doesn't directly compute the honest server's share of $[\Delta \cdot A_i]$ for each opened value A_i. Instead, he computes $\Delta \cdot A_i$ and computes the honest server's share of $[\Delta \cdot A_i]$ based on the corrupted servers' shares and $\Delta \cdot A_i$. We only change the way of generating each of the honest server's shares of $[\Delta \cdot A_i]$ without changing its value, which doesn't change the output distribution. Thus, $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same output distribution.

Note that the honest server's shares of $[\Delta \cdot \Delta_w], [\Delta \cdot \Delta'_w], [\Delta \cdot \Delta_w \cdot \Delta'_w], [\![a_w]\!], [\![b_w]\!], [\![c_w]\!]$ for each input wire w are not used in the later simulation. $\mathcal{S}$ does not compute them in future hybrids.

$\mathbf{Hyb}_6$: In this hybrid, for the execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ while preparing separate MAC keys, after following the protocol to check whether $\sum_{i=1}^{n}[\sigma] = 0$, $\mathcal{S}$ aborts the simulation if $\mathsf{MACCheck} = 1$. The only difference between $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ is that the corrupted servers 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 assume that the honest server is S_j. 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 servers is with an additive error δ_σ.

Then

$$\begin{aligned}\sigma = &\sum_{i\in[1,n],i\neq j}\left(\sum_{k=1}^{m}\chi_k\cdot[\Delta\cdot A_k]-\Delta^{(i)}\cdot\sum_{k=1}^{m}\chi_k\cdot A_k\right)+\delta_\sigma\\ &+\left(\sum_{k=1}^{m}\chi_k\cdot(\Delta\cdot A_k)^{(j)}-\Delta^{(j)}\cdot\sum_{k=1}^{m}\chi_k\cdot(A_k+\delta_k)\right)\\ =&\delta_\sigma+\Delta^{(j)}\cdot\sum_{k=1}^{m}\chi_k\cdot\delta_k.\end{aligned}$$

If some $\delta_k\neq 0$, since the seed that is expanded to $\chi_1,\ldots,\chi_m$ is sampled after the errors are fixed, and there is only a negligible probability of $1/p$ that $\sum_{k=1}^{m}\chi_k\cdot\delta_k=0$ if each χ_k is truly random. Thus, if there is a non-negligible probability that $\sum_{k=1}^{m}\chi_k\cdot\delta_k=0$, then the truly random field elements and the pseudo-random values $\{\chi_k\}_{k=1}^{m}$ can be distinguished by computing $\sum_{k=1}^{m}\chi_k\cdot\delta_k=0$ with a non-negligible probability, which contradicts the definition of a PRG, so there probability that $\sum_{k=1}^{m}\chi_k\cdot\delta_k=0$ is negligible. If $\sum_{k=1}^{m}\chi_k\cdot\delta_k\neq 0$, and the certification passes with a non-negligible probability, then the adversary can compute $\Delta^{(j)}$ with the errors and obtains Δ with a non-negligible probability. Since Δ isn't used in the simulation process to compute any transcript sent to any corrupted server before the corrupted servers commit their shares of $[\sigma]$ and all these transcripts can be generated by g^{Δ}, we can construct a PPT algorithm that runs the previous steps of $\mathcal{S}$ to generates the transcripts sent to $\mathcal{A}$ before the corrupted servers commit their shares of $[\sigma]$ and then runs $\mathcal{A}$ to computes Δ from g^{Δ} with a non-negligible probability, which contradicts the DDH assumption. Therefore, the distribution only changes with negligible probability.

Thus, the distributions of $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ are computationally indistinguishable.

In addition, Δ_w,Δ'_w for input wire w attached to an honest client are not used when simulating Steps 3 and 4.(a). $\mathcal{S}$ delays the sampling of Δ_w,Δ'_w until Step 4.(b).

$\mathbf{Hyb}_7$: In this hybrid, for each honest client C_i and each input wire w attached to client C_i, $\mathcal{S}$ doesn't follow the protocol to check λ_w. Instead, $\mathcal{S}$ checks whether the sum of their shares of $[\lambda_w],[\Delta_w],[\Delta'_w],[\lambda_w\cdot\Delta_w],[\Delta_w\cdot\Delta'_w]$ are all correct. This only changes the distribution if the corrupted servers apply additive errors $\delta_1,\delta_2,\delta_3,\delta_4,\delta_5$ (not all 0) on $\lambda_w,\Delta_w,\Delta'_w,\lambda_w\cdot\Delta_w,\Delta_w\cdot\Delta'_w$ respectively such that $(\lambda_w+\delta_1)\cdot(\Delta_w+\delta_2)=\lambda_w\cdot\Delta_w+\delta_4$ and $(\Delta_w+\delta_2)\cdot(\Delta'_w+\delta_3)=\Delta_w\cdot\Delta'_w+\delta_5$. Then it holds that $\lambda_w\cdot\delta_2+\delta_1\cdot\Delta_w+\delta_1\cdot\delta_2=\delta_4$ and $\Delta_w\cdot\delta_3+\delta_2\cdot\Delta'_w+\delta_2\cdot\delta_3=\delta_5$.

If $\delta_4\neq 0$, then one of δ_1,δ_2 should be non-zero. Similarly, $\delta_5\neq 0$, then one of δ_2,δ_3 should be non-zero. Thus, we only need to consider the case if one of $\delta_1,\delta_2,\delta_3$ is non-zero. If $\delta_1\neq 0$, then the output distribution only changes when

$$\Delta_w=\frac{\delta_4-\lambda_w\cdot\delta_2}{\delta_1}-\delta_2.$$

Since Δ_w is sampled after $\delta_1,\delta_2,\delta_4$ are determined. Thus, there is only 1 element in $\mathbb{F}_p$ can be the value of Δ_w to make $(\lambda_w+\delta_1)\cdot(\Delta_w+\delta_2)=\lambda_w\cdot\Delta_w+\delta_4$ hold, which is with a negligible probability $1/p$. Similarly, if $\delta_2\neq 0$, then the output distribution only changes when

$$\Delta'_w=\frac{\delta_5-\Delta_w\cdot\delta_3}{\delta_2}-\delta_3,$$

and it is with a negligible probability. For the same reason, if $\delta_3\neq 0$, then the output distribution also changes with a negligible probability.

Thus, the distributions of $\mathbf{Hyb}_7$ and $\mathbf{Hyb}_6$ are statistically close.

$\mathbf{Hyb}_8$: In this hybrid, for each input wire w attached to a corrupted client, $\mathcal{S}$ doesn't directly sample the honest server S_j's share of $[\lambda_w]$. Instead, $\mathcal{S}$ samples λ_w first and then computes the honest server S_j's share of $[\lambda_w]$ based on λ_w and the corrupted servers' shares. Since both λ_w and the honest server S_j's share of $[\lambda_w]$ are also uniformly random, this doesn't change the distribution. Thus, $\mathbf{Hyb}_8$ and $\mathbf{Hyb}_7$ have the same output distribution.

$\mathbf{Hyb}_9$: In this hybrid, for each input wire w attached to a corrupted client, $\mathcal{S}$ doesn't follow the protocol to compute the honest server P_j's share of $[\lambda_w \cdot \Delta_w]$ and directly generate P_j's share of $[\Delta_w \cdot \Delta'_w]$. Instead, $\mathcal{S}$ reconstructs Δ_w, Δ'_w with all the servers' shares and computes $\lambda_w \cdot \Delta_w, \Delta_w \cdot \Delta'_w$. Then $\mathcal{S}$ computes P_j's shares of $[\lambda_w \cdot \Delta_w]$ and $[\Delta_w \cdot \Delta'_w]$ based on the secrets and the corrupted servers' shares. We only change order of generating $\lambda_w \cdot \Delta_w, \Delta_w \cdot \Delta'_w$ and P_j's shares of $[\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ without changing their distributions. Thus, $\mathbf{Hyb}_9$ and $\mathbf{Hyb}_8$ have the same output distribution.

$\mathbf{Hyb}_{10}$: In this hybrid, while sending input wire values, for each wire w to an honest client, $\mathcal{S}$ doesn't follow the protocol to compute $v_w \oplus \lambda_w$. Instead, $\mathcal{S}$ samples a random bit as $v_w \oplus \lambda_w$. Then $\mathcal{S}$ computes λ_w by $(v_w \oplus \lambda_w) \oplus v_w$ for these wires. Since λ_w is sampled randomly in $\{0,1\}$, so is $v_w \oplus \lambda_w$. Therefore, we only change the order of generating λ_w and $v_w \oplus \lambda_w$ for these wires without changing their distributions. Thus, $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ have the same output distribution.

Note that the honest server's shares of $[\Delta_w], [\Delta'_w], [\Delta_w \cdot \Delta'_w]$ for each input wire w attached to an honest client are not used in the later simulation. $\mathcal{S}$ does not compute them in future hybrids.

$\mathbf{Hyb}_{11}$: In this hybrid, for each input wire w attached to a corrupted client, after receiving $v_w \oplus \lambda_w$ from the corrupted client, $\mathcal{S}$ additionally computes the corrupted client's input v_w by $v_w \oplus \lambda_w \oplus \lambda_w$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{11}$ and $\mathbf{Hyb}_{10}$ have the same output distribution.

$\mathbf{Hyb}_{12}$: In this hybrid, $\mathcal{S}$ uses the wire values of all the input wires to compute the wire value v_w for each wire w in the circuit. Besides, for each wire w except input wires, $\mathcal{S}$ samples a random bit as $v_w \oplus \lambda_w$ and computes λ_w by $(v_w \oplus \lambda_w) \oplus v_w$ for these wires. Since λ_w is sampled randomly in $\{0,1\}$, so is $v_w \oplus \lambda_w$. The additional computation doesn't affect the output distribution, and we only change the order of generating λ_w and $v_w \oplus \lambda_w$ for these wires without changing their distributions. Thus, $\mathbf{Hyb}_{12}$ and $\mathbf{Hyb}_{11}$ have the same output distribution.

$\mathbf{Hyb}_{13}$: In this hybrid, for each wire w, $\mathcal{S}$ doesn't follow the protocol to compute the honest server S_j's share of $g^{[k_{w,0}]}$. Instead, $\mathcal{S}$ samples a random element in $\mathbb{F}_p$ as the honest server S_j's share $k^{(j)}_{w,v_w \oplus \lambda_w}$ of $[k_{w,v_w \oplus \lambda_w}]$ and then computes $g^{k^{(j)}_{w,v_w \oplus \lambda_w}}$. Then, $\mathcal{S}$ computes the honest server S_j's share $g^{k^{(j)}_{w,0}} = g^{k^{(j)}_{w,v_w \oplus \lambda_w}} \cdot (g^{\Delta^{(j)}})^{-(v_w \oplus \lambda_w)}$ and $k^{(j)}_{w,0} = k^{(j)}_{w,v_w \oplus \lambda_w} - \Delta^{(j)} \cdot (v_w \oplus \lambda_w)$. Finally, $\mathcal{S}$ computes the honest server S_j's share of $[\Delta \cdot k_{w,0}]$. If $v_w \oplus \lambda_w = 0$, this actually makes no difference. If $v_w \oplus \lambda_w = 1$, we just change the order of generating $k^{(j)}_{w,0}$ and $k^{(j)}_{w,1}$ and use $g^{k^{(j)}_{w,0}} = g^{k^{(j)}_{w,1}} \cdot g^{-\Delta^{(j)}}$ to compute $g^{k^{(j)}_{w,0}}$. This doesn't change the distribution of $g^{k^{(j)}_{w,0}}$. Thus, $\mathbf{Hyb}_{13}$ and $\mathbf{Hyb}_{12}$ have the same output distribution.

$\mathbf{Hyb}_{14}$: In this hybrid, for each wire w, if $g^{k_{w,0}}$ is not correctly sent, $\mathcal{S}$ additionally sets $\mathsf{pkCheck} = 1$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{13}$ have the same output distribution.

$\mathbf{Hyb}_{15}$: In this hybrid, if g^r is not correctly sent, $\mathcal{S}$ additionally sets $\mathsf{pkCheck} = 1$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$ have the same output distribution.

$\mathbf{Hyb}_{16}$: In this hybrid, if τ is not correctly sent, $\mathcal{S}$ additionally sets $\mathsf{MACCheck} = 1$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{16}$ and $\mathbf{Hyb}_{15}$ have the same output distribution.

$\mathbf{Hyb}_{17}$: In this hybrid, after following the protocol to check whether $g^\tau = \prod_{i=1}^{W} (g^{k_{w_i,0}})^{\theta_i} \cdot g^r$, if $\mathsf{pkCheck} = 1$ and $\mathsf{MACCheck} = 0$, $\mathcal{S}$ aborts the simulation. This happens only when τ is opened correctly but g^r or $g^{k_{w,0}}$ for some wire w is not correctly opened. Note that the seed that expanded to $\theta_1, \ldots, \theta_W$ isn't sampled before S_1 sends g^r and $g^{k_{w,0}}$. Assume g^r is opened to be $c_r \cdot g^r$ and each $g^{k_{w_k,0}}$ is opened to be $c_k \cdot g^{k_{w_k,0}}$ for each $k \in [1, W]$, where one of c_r and $\{c_k\}_{k=1}^{W}$ is not 1. Then the distribution changes if and only if $c_r \cdot \prod_{i=1}^{W} c_k^{\theta_k} = 1$. Assume that $\theta_1, \ldots, \theta_W$ are uniformly random field elements, then at least for one $k \in [1, W]$, $c_k \neq 1$, there is only one $\theta_k \in \mathbb{F}_p$ that satisfies this equation when other elements are fixed. This happens with a negligible probability $1/p$. If for $\theta_1, \ldots, \theta_W$ expanded by the seed $c_r \cdot \prod_{i=1}^{W} c_k^{\theta_k} = 1$ happens with a non-negligible probability, then we can construct a PPT algorithm that runs the previous steps of $\mathcal{S}$ to compute whether $c_r \cdot \prod_{i=1}^{W} c_k^{\theta_k} = 1$ to distinguish whether the equation is computed by pseudo-random or truly random elements $\theta_1, \ldots, \theta_W$. This contradicts the definition of a PRG. Therefore, the distribution only changes with a negligible probability. Thus, the distributions of $\mathbf{Hyb}_{17}$ and $\mathbf{Hyb}_{16}$ are computationally indistinguishable.

$\mathbf{Hyb}_{18}$: In this hybrid, $\mathcal{S}$ doesn't sample the seed that expanded to $\theta_1, \ldots, \theta_W$ while emulating $\mathcal{F}_{\mathsf{Coin}}$.

Instead, $\mathcal{S}$ samples it earlier at the beginning of **Sending Public Keys**. Since $\theta_1, \ldots, \theta_W$ are not used in computing any transcript that is sent to the corrupted servers, we don't change the output distribution. Thus, $\mathbf{Hyb}_{18}$ and $\mathbf{Hyb}_{17}$ have the same output distribution.

$\mathbf{Hyb}_{19}$: In this hybrid, $\mathcal{S}$ doesn't follow the protocol to compute the honest server S_j's share of $[\![\tau]\!]$. Instead, $\mathcal{S}$ samples a random element in $\mathbb{F}_p$ at the beginning of **Sending Public Keys** as τ. $\mathcal{S}$ computes the honest server S_j's share of $[\![\tau]\!]$ based on the corrupted servers' shares, τ, and Δ. Then, $\mathcal{S}$ computes $g^{r^{(j)}}$ based on $g^{\tau^{(j)}} = \prod_{i=1}^{W}(g^{k_{w_i,0}^{(j)}})^{\theta_i} \cdot g^{r^{(j)}}$ and $\tau^{(j)}$. $\mathcal{S}$ also computes S_j's shares of $[\![r]\!]$ by $[\![\tau]\!] - \sum_{i=1}^{W} \theta_i \cdot [\![k_{w_i,0}]\!]$. Since r is uniformly random, so is τ. Hence, we only change the order of generating the honest server S_j's share of $[\![\tau]\!]$ and $[\![r]\!]$ without changing their distributions. Thus, $\mathbf{Hyb}_{19}$ and $\mathbf{Hyb}_{18}$ have the same output distribution.

Note that $[\![r]\!]$ is not used in the later simulation. $\mathcal{S}$ does not compute S_j's shares of $[\![r]\!]$ in future hybrids. In addition, S_j's shares of $\{[\![k_{w,0}]\!]\}_{i=1}^{W}$ are not used when simulating Step 7. $\mathcal{S}$ delays the computation of S_j's shares of $\{[\![k_{w,0}]\!]\}_{i=1}^{W}$ until Step 8.(c).

$\mathbf{Hyb}_{20}$: In this hybrid, while computing sharings of output labels, for each execution of Π_{Mult}, $\mathcal{S}$ doesn't follow the protocol to compute the honest server S_j's shares of $[e]$ and $[d]$. Instead, $\mathcal{S}$ samples two random elements in $\mathbb{F}_p$ as the honest server S_j's shares of $[e]$ and $[d]$ and then computes the corresponding random shares $[a], [b]$ for the triple used in this multiplication. Then $\mathcal{S}$ reconstructs a, b and computes the honest parties' shares of $[\Delta \cdot a], [\Delta \cdot b], [\![c]\!]$ correspondingly with a, b and corrupted parties' shares. For the same reason in $\mathbf{Hyb}_2$, $\mathbf{Hyb}_{20}$ and $\mathbf{Hyb}_{19}$ have the same output distribution.

$\mathbf{Hyb}_{21}$: In this hybrid, while computing sharings of output labels, for each execution of Π_{Mult}, if either of e, d is not sent correctly, $\mathcal{S}$ additional sets $\mathsf{MACCheck} = 1$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{21}$ and $\mathbf{Hyb}_{20}$ have the same output distribution.

$\mathbf{Hyb}_{22}$: In this hybrid, while simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, $\mathcal{S}$ additionally sets $\mathsf{MACCheck} = 1$ if the sum of the shares of $[\sigma]$ committed by corrupted servers is not correct. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{22}$ and $\mathbf{Hyb}_{21}$ have the same output distribution.

$\mathbf{Hyb}_{23}$: In this hybrid, if $\mathsf{MACCheck} = 0$, while simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, $\mathcal{S}$ doesn't compute the honest server's share of $[\sigma]$ by himself. Instead, $\mathcal{S}$ computes the honest server's share of $[\sigma]$ with the corrupted servers' shares and $\sum_{i=1}^{n}[\sigma] = 0$. Since $\mathsf{MACCheck} = 0$, each pair of e, d is opened correctly, which guarantees $\sum_{i=1}^{n}[\sigma] = 0$, so we only change the way of generating the honest server's share of $[\sigma]$ without changing its distribution. Thus, $\mathbf{Hyb}_{23}$ and $\mathbf{Hyb}_{22}$ have the same output distribution.

Note that in this case the honest server S_j's share of $[\Delta \cdot \tau]$ is not used. $\mathcal{S}$ does not generate S_j's share of $[\Delta \cdot \tau]$ in future hybrids.

$\mathbf{Hyb}_{24}$: In this hybrid, if $\mathsf{MACCheck} = 1$, while simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, $\mathcal{S}$ doesn't directly compute the honest server's share of $[\Delta \cdot A_i]$ for each opened value A_i. Instead, he computes $\Delta \cdot A_i$ and computes the honest server's share of $[\Delta \cdot A_i]$ based on the corrupted servers' shares and $\Delta \cdot A_i$. We only change the way of generating each of the honest server's shares of $[\Delta \cdot A_i]$ without changing its value, which doesn't change the output distribution. Thus, $\mathbf{Hyb}_{24}$ and $\mathbf{Hyb}_{23}$ have the same output distribution.

Note that the honest server's shares of each triple used in step 8.(a) are not used in the later simulation. $\mathcal{S}$ does not compute them in future hybrids. Also the MAC sharing of each λ_w, $[\Delta \cdot \lambda_w]$ is not used until Step 8.(c), $\mathcal{S}$ delays the generation of the honest server's share of $[\Delta \cdot \lambda_w]$ for all λ_w.

$\mathbf{Hyb}_{25}$: In this hybrid, while simulating the second execution of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, after following the protocol to check whether $\sum_{i=1}^{n}[\sigma] = 0$, $\mathcal{S}$ aborts the simulation if $\mathsf{MACCheck} = 1$. For the same reason in $\mathbf{Hyb}_6$, the distributions of $\mathbf{Hyb}_{25}$ and $\mathbf{Hyb}_{24}$ are computationally indistinguishable.

$\mathbf{Hyb}_{26}$: In this hybrid, for each gate g with input wires a, b and output wire c, $\mathcal{S}$ doesn't follow the protocol to compute the honest server S_j's shares of $[\![\chi_1]\!], [\![\chi_2]\!], [\![\chi_3]\!], [\![\chi_4]\!]$. Instead, $\mathcal{S}$ first computes χ_i using $\lambda_a, \lambda_b, \lambda_c$ and then generates S_j's shares of $[\![\chi_i]\!]$ based on χ_i, Δ, and shares of corrupted parties for each $i = 1, 2, 3, 4$. Given that the check in Step 8.(b) does not abort, corrupted parties follow the protocol when computing $[\![\chi_i]\!]$. Thus, the distributions of $\mathbf{Hyb}_{26}$ and $\mathbf{Hyb}_{25}$ are identical.

Note that the honest server S_j's share of $[\Delta \cdot \lambda_w]$ for each wire w is not used. $\mathcal{S}$ does not generate these shares in future hybrids.

$\mathbf{Hyb}_{27}$: In this hybrid, for each gate g with input wires a, b and output wire c, let $i^\star = 2(v_a \oplus \lambda_a) +$

$(v_b \oplus \lambda_b) + 1$. $\mathcal{S}$ doesn't follow the protocol to compute the honest server S_j's share of $[x_{c,i^\star}]$. Instead, $\mathcal{S}$ computes it based on the secret $x_{c,i^\star} = k_{c,v_c \oplus \lambda_c}$ and corrupted servers' shares of $[x_{c,i^\star}]$. By construction, we have $v_c = f_g(v_a, v_b)$. Thus, $\chi_{i^\star} = f_g(v_a, v_b) \oplus \lambda_c = v_c \oplus \lambda_c$, and $x_{c,i^\star} = k_{c,0} + \chi_{i^\star} \cdot \Delta = k_{c,v_c \oplus \lambda_c}$. This does not change the distribution of S_j's share of $[x_{c,i^\star}]$. Thus, the distributions of $\mathbf{Hyb}_{27}$ and $\mathbf{Hyb}_{26}$ are identical.

$\mathbf{Hyb}_{28}$: In this hybrid, $\mathcal{S}$ maintains a set Q. For each gate g with input wires a, b and output wire c, and for all $i_0, i_1 \in \{0, 1\}$ such that $(i_0, i_1) \neq (v_a \oplus \lambda_a, v_b \oplus \lambda_b)$, when the honest server S_j computes $\mathsf{Enc}(pp, g^{k_{a,i_0}}, g^{k_{b,i_1}}, x^{(j)}_{c,2i_0+i_1+1})$, $\mathcal{S}$ checks whether the query to the random oracle has been queried before. If true, $\mathcal{S}$ aborts the simulation. Otherwise, $\mathcal{S}$ adds the query to Q. Note that when $\Delta \neq 0$, which happens with overwhelming probability, each query made to the random oracle is uniformly random. Thus, the probability that some query has been queried (either by the honest server or by the adversary) is negligible. The distributions of $\mathbf{Hyb}_{28}$ and $\mathbf{Hyb}_{27}$ are computationally indistinguishable.

$\mathbf{Hyb}_{29}$: In this hybrid, while doing encryption for each cipher-text except $\mathsf{Enc}(pp, g^{k_{a,v_a \oplus \lambda_a}}, g^{k_{b,v_b \oplus \lambda_b}}, x^{(j)}_{c,v_c \oplus \lambda_c})$, $\mathcal{S}$ doesn't follow the encryption algorithm to query the random oracle and then use the result to encrypt the message. Instead, for message m, $\mathcal{S}$ directly samples m^*, k_1, k_2 randomly (where $m^* = m \oplus \mathcal{O}((\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2})$ is one element of the cipher-text), and sets the output of the query $(\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2}$ to the random oracle to be $m^* \oplus m$. Note that the only difference between $\mathbf{Hyb}_{28}$ and $\mathbf{Hyb}_{29}$ is the way we decide the output for queries in Q. Since m^* is randomly sampled, $m^* \oplus m$ is also uniformly random. In particular, when $\mathcal{S}$ does not abort the simulation, queries in Q have not been queried before. Thus, the distributions of $\mathbf{Hyb}_{29}$ and $\mathbf{Hyb}_{28}$ are identical.

$\mathbf{Hyb}_{30}$: In this hybrid, while doing encryption for each cipher-text except $\mathsf{Enc}(pp, g^{k_{a,v_a \oplus \lambda_a}}, g^{k_{b,v_b \oplus \lambda_b}}, x^{(j)}_{c,v_c \oplus \lambda_c})$, $\mathcal{S}$ does not compute the original message m and does not set the output of the query $(\mathsf{pk}_1)^{k_1} \cdot (\mathsf{pk}_2)^{k_2}$ to the random oracle to be $m^* \oplus m$. Instead, $\mathcal{S}$ honestly emulates the random oracle. In particular, $\mathcal{S}$ no longer checks whether the query to the random oracle when S_j computing the cipher-texts has been queried before. We prove that the distributions of $\mathbf{Hyb}_{30}$ and $\mathbf{Hyb}_{29}$ are computationally indistinguishable.

For the sake of contradiction, assume that there exists an adversary $\mathcal{A}$ such that $\mathbf{Hyb}_{29}$ and $\mathbf{Hyb}_{30}$ are computationally distinguishable. We first construct another two hybrids $\mathbf{Hyb}'_{29}$ and $\mathbf{Hyb}'_{30}$, which are the same as $\mathbf{Hyb}_{29}$ and $\mathbf{Hyb}_{30}$ except that when $\mathcal{S}$ marks $\mathsf{MACCheck} = 1$, $\mathcal{S}$ aborts the simulation. We argue that $\mathbf{Hyb}'_{29}$ and $\mathbf{Hyb}'_{30}$ are also computationally distinguishable. This is because $\mathcal{S}$ only marks $\mathsf{MACCheck} = 1$ before Step 8.(b) and once $\mathsf{MACCheck} = 1$, either $\mathcal{S}$ will abort the simulation or $\mathcal{S}$ will abort the computation on behalf of the honest server at the end of Step 8.(b). On the other hand, $\mathbf{Hyb}_{29}$ and $\mathbf{Hyb}_{30}$ are only different in Step 8.(c). Thus $\mathbf{Hyb}'_{29}$ and $\mathbf{Hyb}'_{30}$ can be distinguished with the same advantage as $\mathbf{Hyb}_{29}$ and $\mathbf{Hyb}_{30}$.

Let Q be the set of queries to the random oracle when S_j computes his cipher-text in $\mathbf{Hyb}'_{30}$ except $\mathsf{Enc}(pp, g^{k_{a,v_a \oplus \lambda_a}}, g^{k_{b,v_b \oplus \lambda_b}}, x^{(j)}_{c,v_c \oplus \lambda_c})$. Now we argue that, with non-negligible probability, at least one query in Q has been queried. Suppose this is not the case. By the same analysis, with overwhelming probability, all queries in Q are distinct. Then by assumption, with overwhelming probability, no query in Q has been queried and all queries in Q are distinct. In this case, the only difference between $\mathbf{Hyb}'_{29}$ and $\mathbf{Hyb}'_{30}$ is that we do not explicitly compute the output to each query in Q by $m^* \oplus m$. Since no query in Q has been queried, this makes no difference in the output distribution. Then it shows that $\mathbf{Hyb}'_{29}$ and $\mathbf{Hyb}'_{30}$ are computationally indistinguishable, which leads to a contradiction.

Thus, with non-negligible probability, at least one query in Q has been queried in $\mathbf{Hyb}'_{30}$. We will use such an adversary $\mathcal{A}$ to break the DDH assumption. First note that $\mathbf{Hyb}'_{30}$ can be generated by only using g^Δ but not Δ. Let T be the upper bound of the number of queries to the random oracle (which depends on the running time of $\mathcal{A}$ and is upper bounded by a polynomial in the security parameter). Note that the size of Q is $3G_A + 3G_X$. We construct an attacker B as follows.

1. B receives (g^Δ, g^r, h) from the challenger, where h is either a random group element or $h = g^{\Delta \cdot r}$.
2. B randomly samples $i_0 \in \{1, \ldots, T\}$ and $i_1 \in \{1, \ldots, 3G_A + 3G_X\}$, representing that B guesses that the i_0-th query to the random oracle is the i_1-th query in Q.

3. B invokes $\mathcal{S}$ and uses g^{Δ} to execute with $\mathcal{A}$ in the same way as $\mathbf{Hyb}'_{30}$ except of preparing the cipher-text corresponding to the i_1-th query in Q:

   - If this query is in the form of $(g^{k_{a,v_a\oplus\lambda_a}})^{k_1}\cdot(g^{k_{b,1\oplus v_b\oplus\lambda_b}})^{k_2}$, B randomly samples k_1, computes g^{k_1}, and sets $g^{k_2}=g^r$. After learning the i_0-th query q to the random oracle, B checks whether
   $$q=(g^{k_{a,v_a\oplus\lambda_a}})^{k_1}\cdot(g^r)^{k_{b,v_b\oplus\lambda_b}}\cdot h.$$
   - If this query is in the form of $(g^{k_{a,1\oplus v_a\oplus\lambda_a}})^{k_1}\cdot(g^{k_{b,v_b\oplus\lambda_b}})^{k_2}$, B randomly samples k_2, computes g^{k_2}, and sets $g^{k_1}=g^r$. After learning the i_0-th query q to the random oracle, B checks whether
   $$q=(g^r)^{k_{a,v_a\oplus\lambda_a}}\cdot(g^{k_2})^{k_{b,v_b\oplus\lambda_b}}\cdot h.$$
   - If this query is in the form of $(g^{k_{a,1\oplus v_a\oplus\lambda_a}})^{k_1}\cdot(g^{k_{b,1\oplus v_b\oplus\lambda_b}})^{k_2}$, B randomly samples k_1, computes g^{k_1}, and sets $g^{k_2}=g^r/g^{k_1}$. After learning the i_0-th query q to the random oracle, B checks whether
   $$q=(g^{k_1})^{k_{a,v_a\oplus\lambda_a}}\cdot(g^{k_2})^{k_{b,v_b\oplus\lambda_b}}\cdot h.$$

   If true, B outputs 1. Otherwise, B outputs a random bit.

We show that B has a non-negligible advantage to distinguish $g^{\Delta\cdot r}$ from a random group element. First note that if h is a random group element, then with overwhelming probability, B outputs a random bit since h is not used when interacting with $\mathcal{A}$. When $h=g^{\Delta\cdot r}$, B will always output 1 when the i_0-th query to the random oracle is the i_1-th query in Q, which happens with non-negligible probability. Thus, B has a non-negligible advantage of distinguishing $g^{\Delta\cdot r}$ from a random group element, which contradicts with the DDH assumption.

In summary, $\mathbf{Hyb}_{29}$ and $\mathbf{Hyb}_{30}$ are computationally indistinguishable. Note that the honest server S_j's shares $[\![\chi_1]\!],\ldots,[\![\chi_4]\!]$ for each gate are not used. $\mathcal{S}$ no longer generates those shares in future hybrids. Also, for each input wire associated with an honest client and for each intermediate wire, $\mathcal{S}$ does not generate the honest server S_j's share of $[\lambda_w]$, which is not used anymore.

$\mathbf{Hyb}_{31}$: In this hybrid, $\mathcal{S}$ computes $v_w=(v_w\oplus\lambda_w)\oplus\lambda_w$, sends the corrupted clients' inputs to $\mathcal{F}$, and gets their outputs, and honest clients still compute their outputs by themselves. For each output wire w associated with a corrupted client, when computing λ_w from $v_w\oplus\lambda_w$ and v_w, $\mathcal{S}$ uses the outputs received from $\mathcal{F}$. This doesn't affect the output distribution. Thus, the distribution of $\mathbf{Hyb}_{31}$ and $\mathbf{Hyb}_{30}$ are the same.

$\mathbf{Hyb}_{32}$: In this hybrid, for each honest client C_i and each output wire w attached to client C_i, $\mathcal{S}$ doesn't follow the protocol to check λ_w. Instead, $\mathcal{S}$ checks whether the sum of their shares of $[\lambda_w],[\Delta_w],[\Delta'_w],[\lambda_w\cdot\Delta_w],[\Delta_w\cdot\Delta'_w]$ are all correct. For the same reason in $\mathbf{Hyb}_8$, the distributions of $\mathbf{Hyb}_{32}$ and $\mathbf{Hyb}_{31}$ are statistically close.

$\mathbf{Hyb}_{33}$: In this hybrid, for each output wire w attached to a corrupted client, $\mathcal{S}$ doesn't follow the protocol to compute the honest server P_j's share of $[\lambda_w\cdot\Delta_w]$ and directly generate P_j's share of $[\Delta_w\cdot\Delta'_w]$. Instead, $\mathcal{S}$ reconstructs Δ_w,Δ'_w with all the servers' shares and computes $\lambda_w\cdot\Delta_w,\Delta_w\cdot\Delta'_w$. Then $\mathcal{S}$ computes P_j's shares of $[\lambda_w\cdot\Delta_w]$ and $[\Delta_w\cdot\Delta'_w]$ based on the secrets and the corrupted servers' shares. We only change order of generating $\lambda_w\cdot\Delta_w,\Delta_w\cdot\Delta'_w$ and P_j's shares of $[\lambda_w\cdot\Delta_w],[\Delta_w\cdot\Delta'_w]$ without changing their distributions. Thus, $\mathbf{Hyb}_{33}$ and $\mathbf{Hyb}_{32}$ have the same output distribution.

$\mathbf{Hyb}_{34}$: In this hybrid, honest clients get their outputs from $\mathcal{F}$ instead of computing them by themselves. $\mathcal{S}$ does not compute all wire values in the circuit and does not generate the honest server S_j's share of $[\lambda_w]$ for each wire w associated with an honest client. Since the computation processes of v_w of each output wire w by $\mathcal{S}$ and by $\mathcal{F}$ are the same, the distribution only changes when S_1 correctly sends $v_w\oplus\lambda_w\oplus 1$ and $k_{w,v_w\oplus\lambda_w\oplus 1}$ to an honest client with the output wire w attached to him. Since $|k_{w,v_w\oplus\lambda_w\oplus 1}-k_{w,v_w\oplus\lambda_w}|=|\Delta|$ and $k_{w,v_w\oplus\lambda_w}$ is explicitly generated by $\mathcal{S}$, if $\mathbf{Hyb}_{34}$ and $\mathbf{Hyb}_{33}$ are computationally indistinguishable, then we may construct a PPT algorithm to compute Δ from g^{Δ} with a non-negligible probability, which breaks the DDH assumption. Thus, the distributions of $\mathbf{Hyb}_{34}$ and $\mathbf{Hyb}_{33}$ are computationally indistinguishable.

Note that $\mathbf{Hyb}_{34}$ is the ideal-world scenario, Π_{main} computes $\mathcal{F}$ with computational security.

□

F Cost Analysis for the Main Protocol

We first analyze the communication complexity of Π_{main} in the hybrid model, without considering the cost of communication with functionalities.

During the garbling phase, the communication in each step is as follows:

1. **Initialization.** No communication.
2. **Preparing Mask Sharings.** No communication.
3. **Preparing Separate MAC Keys.** For each input and output wire, the servers run Π_{Mult} once, which requires the servers to open $2(W_I+W_O)$ field elements. This requires communication of $8n(W_I+W_O)\lambda$ bits.
4. **Revealing Input Masks.** Each server needs to send his shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ to a single client for each input wire w, which requires $10nW_I\lambda$ bits of communication.
5. **Sending Input Wire Values.** For each input wire w attached to a client C_i, C_i needs to send a bit $v_w \oplus \lambda_w$ to all the servers, which requires communication of $2nW_I$ bits in total.
6. **Preparing Shares of Wire Labels.** No communication.
7. **Sending Public Keys.** For each wire, each server needs to send a group element to and receive another group element from S_1, which requires communication of $8n\lambda$ bits. The communication cost of sending shares of g^r and τ is independent of the number of wires, so we omit the cost of this part. Thus, the total communication in this step is $8nW\lambda$ bits.
8. **Garbling the Circuit.** For each AND gate, the servers need to run Π_{Mult} 4 times, which requires the servers to open 8 field elements through S_1. This requires communication of $32nG_A\lambda$ bits in total. Similarly, for each XOR gate, the servers only need to run Π_{Mult} twice, which requires $8nG_X\lambda$ bits of communication in total. Using the public key encryption scheme PKE_2, the size of each ciphertext is about 5λ. Thus, sending the ciphertexts requires communication of $40n(G_A+G_X)\lambda$ bits in total. This brings the total communication in this step to $(72G_A + 56G_X)n\lambda$ bits.
9. **Verification of MACs.** During the execution of $\Pi_{\mathsf{SPDZ-MAC}}$, the servers only do local computation and communicate with ideal functionalities $\mathcal{F}_{\mathsf{Coin}}$ and $\mathcal{F}_{\mathsf{Commit}}$, so there is no communication in this step.

During the circuit evaluation phase, we analyze the communication of each step as follows:

1. **Revealing Input Labels.** For each input wire, all the servers send a field element to S_1, which requires $2nW_I\lambda$ bits of communication in total.
2. **Computing the Circuit.** No communication.
3. **Sending Outputs.** For each output wire, S_1 sends a bit and a field element to a single client which requires $2W_O(1+\lambda)$-bit communication. Moreover, each server needs to send his shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ to a single client for each input wire w, which requires $6nW_O\lambda$-bit communication. Thus, the total communication of this step is about $10nW_O\lambda$ bits.

In total, the communication cost of Π_{main} is about $(20W_I + 18W_O + 8W + 72G_A + 56G_X)n\lambda$ bits.

If we use Π_{prep} to realize $\mathcal{F}_{\mathsf{prep}}$, the execution of Π_{prep} requires communication of about $(12nm_T + 16nm_B)\lambda = (24W_I + 24W_O + 16W + 48G_A + 24G_X)n\lambda$ bits, resulting in a total communication of about $(44W_I + 42W_O + 24W + 120G_A + 80G_X)n\lambda$ bits. Since $W = G_A + G_X + W_I$, the total communication for the complete protocol is about $(58W_I + 42W_O + 144G_A + 104G_X)n\lambda$ bits.

G Protocol without Random Oracles

In this appendix, we show how to instantiate our protocol without assuming the existence of random oracles. We still use the same preprocessing as our protocol based on random oracles. This time, the public key encryption scheme we used can be $\mathsf{PKE}_1 = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ from Section 4.1. We still provide our main protocol Π_{main} in the $\{\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}\}$-hybrid model. The main protocol Π_{main} runs Π_{Garbling} and Π_{Eval} in order, where Π_{Garbling} and Π_{Eval} are provided as follows.

Protocol Π_{Garbling}

Garbling Phase

Let λ be the computational security parameter and κ be the statistical security parameter. All the servers agree on the public parameter $pp \leftarrow \mathsf{Setup}(\lambda, \kappa)$ from the public-key encryption scheme $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ constructed in Section 4.

1. **Initialization.** Set $m_T = 5G_A + 3G_X + 2W_I + 2W_O$, $m_B = W$, $m_R = 2W + 1$. The servers send $(\mathsf{Init}, m_T, m_R, m_B)$ to $\mathcal{F}_{\mathsf{prep}}$ and receive the outputs from $\mathcal{F}_{\mathsf{prep}}$.
2. **Preparing Mask Sharings.** For each wire w, all the servers take one random sharing of a bit generated by **Random Bits** of $\mathcal{F}_{\mathsf{prep}}$ as $[\![\lambda_w]\!]$. λ_w serves as the mask of the wire value of w.
3. **Preparing Separate MAC Keys.** For each input and output wire w:
   (a) The servers take a triple generated by $\mathcal{F}_{\mathsf{prep}}$ as $[\![\Delta_w]\!], [\![\Delta'_w]\!], [\![\Delta_w \cdot \Delta'_w]\!]$.
   (b) The servers take another triple $[\![a_w]\!], [\![b_w]\!], [\![c_w]\!]$ and run Π_{Mult} on $[\![\Delta_w]\!]$ and $[\![\lambda_w]\!]$ to obtain $[\![\Delta_w \cdot \lambda_w]\!]$.
   (c) The servers run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check the MACs on all the opened values, i.e. $(W_I + W_O)$ pairs of d, e opened in $(W_I + W_O)$ executions of Π_{Mult}.
4. **Revealing Input Masks.** For each input wire attached to each client C_i:
   (a) Each server sends his shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ to C_i.
   (b) C_i reconstructs $\lambda_w, \Delta_w, \Delta'_w, \lambda_w \cdot \Delta_w, \Delta_w \cdot \Delta'_w$. If $\lambda_w \cdot \Delta_w$ is not equal to the product of λ_w and Δ_w, or $\Delta_w \cdot \Delta'_w$ is not equal to the product of Δ_w and Δ'_w, abort the protocol.
5. **Sending Input Wire Values.** For each client C_i and each input wire w attached to C_i, C_i sends $v_w \oplus \lambda_w$ to all the servers, where v_w is the input wire value of w.
6. **Preparing Shares of Wire Labels.** For each wire w, the servers take two random sharings $[\![k_{w,0}]\!], [\![k_{w,1}]\!]$ generated by **Random Values** of $\mathcal{F}_{\mathsf{prep}}$. Let S_i's share of each $[k_{w,b}]$ be $k^{(i)}_{w,b}$ for $b \in \{0, 1\}$.
7. **Sending Public Keys.**
   (a) For each wire w, the servers compute their shares $\{g^{k^{(i)}_{w,0}}, g^{k^{(i)}_{w,1}}\}_{i=1}^n$ of $g^{[k_{w,0}]}, g^{[k_{w,1}]}$ and send them to S_1. S_1 computes and sends $g^{k_{w,0}}, g^{k_{w,1}}$ to all servers.
   (b) The servers use a random sharing $[\![r]\!]$ generated by **Random Values** of $\mathcal{F}_{\mathsf{prep}}$. Then all servers compute their shares of $g^{[r]}$ and send them to S_1. S_1 computes and sends g^r to all the servers.
   (c) The servers call $\mathcal{F}_{\mathsf{Coin}}$ to get a random seed in $\mathbb{F}_p$ and expand it to get random values $\theta_1, \ldots, \theta_{2W} \in \mathbb{F}_p$, and locally compute $[\![\tau]\!] = \sum_{i=1}^W \theta_i \cdot [\![k_{w_i,0}]\!] + \sum_{i=1}^W \theta_{W+i} \cdot [\![k_{w_i,1}]\!] + [\![r]\!]$. Then, the servers run Π_{Open} to open τ.
   (d) Each server checks whether $g^\tau = \prod_{i=1}^W (g^{k_{w_i,0}})^{\theta_i} \cdot \prod_{i=1}^W (g^{k_{w_i,1}})^{\theta_{W+i}} \cdot g^\tau$. If not, abort the protocol.
8. **Garbling the Circuit.** For each gate g with input wires a, b and output wire c, let $f_g : \{0,1\}^2 \to \{0,1\}$ be the function computed by the gate.
   (a) **Computing Sharings of Output Labels.**
      i. All the servers jointly compute SPDZ sharings of

$$\chi_1 = f_g(0 \oplus \lambda_a, 0 \oplus \lambda_b) \oplus \lambda_c, \qquad \chi_2 = f_g(0 \oplus \lambda_a, 1 \oplus \lambda_b) \oplus \lambda_c,$$

$$\chi_3 = f_g(1 \oplus \lambda_a, 0 \oplus \lambda_b) \oplus \lambda_c, \qquad \chi_4 = f_g(1 \oplus \lambda_a, 1 \oplus \lambda_b) \oplus \lambda_c.$$

      Note that $\lambda_w^2 = \lambda_w$ for each wire w.

- For AND gates, servers run Π_{Mult} to compute $[\![\lambda_a \cdot \lambda_b]\!], [\![\lambda_c \cdot \lambda_b]\!], [\![\lambda_a \cdot \lambda_c]\!], [\![\lambda_a \cdot \lambda_b \cdot \lambda_c]\!]$. Note that each χ_j can be viewed as a linear combination of $\{1, \lambda_a, \lambda_b, \lambda_c, \lambda_a \cdot \lambda_b, \lambda_a \cdot \lambda_c, \lambda_b \cdot \lambda_c, \lambda_a \cdot \lambda_b \cdot \lambda_c\}$.
- For XOR gates, note that $\chi_1 = \chi_4 = \lambda_a \oplus \lambda_b \oplus \lambda_c$ and $\chi_2 = \chi_3 = 1 \oplus \lambda_a \oplus \lambda_b \oplus \lambda_c$. All servers run Π_{Mult} to compute $[\![\lambda_a \cdot \lambda_b]\!]$ and then locally compute $[\![\lambda_a \oplus \lambda_b]\!] = [\![\lambda_a]\!] + [\![\lambda_b]\!] - 2 \cdot [\![\lambda_a \cdot \lambda_b]\!]$. Similarly, they get $[\![\lambda_a \oplus \lambda_b \oplus \lambda_c]\!]$ using one call of Π_{Mult}.

ii. All servers locally compute $[\![\chi_j]\!]$ for each $j = 1, 2, 3, 4$ and then compute $[\![k_{c,1} - k_{c,0}]\!]$.

iii. The the servers run Π_{Mult} to compute $[\![(k_{c,1} - k_{c,0}) \cdot \chi_j]\!]$ and then compute $[\![x_{c,j}]\!] = [\![k_{c,0}]\!] + [\![(k_{c,1} - k_{c,0}) \cdot \chi_j]\!]$ for each $j = 1, 2, 3, 4$.

(b) **Verification of MACs.** The servers run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check the MACs on all the opened values, i.e. τ and $5G_A + 3G_X$ pairs of d, e opened in $5G_A + 3G_X$ executions of Π_{Mult}.

(c) **Encrypting Output Labels.** Each server S_i encrypts his share $x_{c,1}^{(i)}$ (of $[x_{c,1}]$) by $\mathsf{Enc}(pp, g^{k_{a,0}}, g^{k_{b,0}}, x_{c,1}^{(i)})$, $x_{c,2}^{(i)}$ by $\mathsf{Enc}(pp, g^{k_{a,0}}, g^{k_{b,1}}, x_{c,2}^{(i)})$, $x_{c,3}^{(i)}$ by $\mathsf{Enc}(pp, g^{k_{a,1}}, g^{k_{b,0}}, x_{c,3}^{(i)})$, and $x_{c,4}^{(i)}$ by $\mathsf{Enc}(pp, g^{k_{a,1}}, g^{k_{b,1}}, x_{c,4}^{(i)})$. Then, S_i sends the ciphertexts to S_1.

Figure 20: Protocol for the garbling phase.

Protocol Π_{Eval}

Circuit Evaluation Phase

1. **Revealing Input Labels.** For each input wire w, all the servers send their shares of $[k_{w,v_w \oplus \lambda_w}]$ to S_1. S_1 checks whether $k_{w,v_w \oplus \lambda_w}$ is consistent with the corresponding public key. If not, abort the protocol.
2. **Computing the Circuit.** S_1 computes the circuit gate by gate. For each gate with input wires a, b and output wire c, if S_1 knows $k_{a,v_a \oplus \lambda_a}, k_{b,v_b \oplus \lambda_b}$, he can use them to decrypt all the servers' shares of $k_{c,v_c \oplus \lambda_c}$. Then S_1 computes $g^{k_{c,v_c \oplus \lambda_c}}$ and compares it with $g^{k_{c,0}}$ and $g^{k_{c,1}}$ to learn $v_c \oplus \lambda_c$. If $g^{k_{c,v_c \oplus \lambda_c}}$ is not in $\{g^{k_{c,0}}, g^{k_{c,1}}\}$, abort the protocol.
3. **Sending Outputs.** For each client C_i and each output wire w attached to C_i:
   (a) S_1 sends $v_w \oplus \lambda_w$ and $k_{w,v_w \oplus \lambda_w}$ to C_i. Then C_i checks whether they match the public key $g^{k_{w,v_w \oplus \lambda_w}}$. If not, abort the protocol.
   (b) Each server sends his shares of $[\lambda_w], [\Delta_w], [\Delta'_w], [\lambda_w \cdot \Delta_w], [\Delta_w \cdot \Delta'_w]$ to C_i.
   (c) C_i reconstructs $\lambda_w, \Delta_w, \Delta'_w, \lambda_w \cdot \Delta_w, \Delta_w \cdot \Delta'_w$. If $\lambda_w \cdot \Delta_w$ is not equal to the product of λ_w and Δ_w, or $\Delta_w \cdot \Delta'_w$ is not equal to the product of Δ_w and Δ'_w, abort the protocol.
   (d) C_i computes his output v_w from $v_w \oplus \lambda_w$ and λ_w.

Figure 21: Protocol for the circuit evaluation phase.

This protocol gives us the following theorem.

Theorem 6. *Assuming DDH and LPN, there is a computationally secure constant-round MPC protocol against a fully malicious adversary controlling up to $n - 1$ parties with communication of $O(|C|n\lambda)$ bits, where λ is the computational security parameter.*