Constant-Round Asynchronous MPC with Optimal Resilience and Linear Communication

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

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

Abstract

In this work, we consider secure multiparty computation (MPC) in the asynchronous network setting. MPC allows n parties to compute a public function on their private inputs against an adversary corrupting at most t of them. We consider both communication complexity and round complexity of asynchronous MPC (AMPC) with the optimal resilience $n = 3t + 1$.

Without fully homomorphic encryptions, the best-known result in this setting is achieved by Coretti, Garay, Hirt, and Zikas (ASIACRYPT 2016), which requires $O(|C|n^3\kappa)$ bits of communication assuming one-way functions, where κ is the security parameter. On the other hand, the best-known non-constant-round AMPC by Goyal, Liu, and Song (CRYPTO 2024) can achieve $O(|C|n)$ communication even in the information-theoretic setting. In this work, we give the first construction of a constant-round AMPC with $O(|C|n\kappa)$ bits of communication that achieves malicious security with abort assuming random oracles. We provide new techniques for adapting the MPC-in-the-head framework in the asynchronous network to compute a constant-size garbled circuit.

Contents

1 Introduction

Secure multiparty computation (MPC) [Yao82, GMW87, BGW88, CCD88, RB89] allows a set of n parties $P_1, \ldots, P_n$ to jointly evaluate a public circuit C on their private inputs against an adversary corrupting at most t parties. The security of MPC protocols guarantees that the information of honest parties' inputs will not be leaked to the adversary beyond what he can learn from corrupted parties' inputs and outputs.

Asynchronous Network Setting. There are two main network setups considered in the literature of MPC: the *synchronous* network setting and the *asynchronous* network setting. So far, most MPC protocols consider the synchronous setting, where there exists a synchronized global clock and an upper bound on the time delay of each message. This allows the parties to run the protocol in rounds, where each message is guaranteed to be delivered to the receiver at the end of each round. This property simplifies the construction of MPC protocols. However, in the real world, such a global clock and upper bound on network latency do not exist. In the asynchronous setting, each message can be arbitrarily delayed and is only guaranteed to be delivered to the receiver eventually. To model the worst case, the adversary is allowed to schedule the delivery of the messages. As a result, when a message is not delivered, an honest receiver cannot distinguish between a corrupted sender not sending a message and an honest sender whose message is delayed by the adversary in the asynchronous setting, which makes the construction of AMPC much more challenging. In particular, it is known that malicious security is possible in the synchronous setting against $t < n/2$ corrupted parties [RB89, Bea90] while in the asynchronous setting, the best possible resilience is $t < n/3$ [BKR94].

Communication and Round Complexity of AMPC. The efficiency of MPC protocols can be measured from various aspects, and the two most common criteria are the communication complexity and the round complexity. The study of the round complexity of AMPC is initialized by Coretti et al. in [CGHZ16]. In particular, they model asynchronous secure channels and asynchronous Byzantine agreements (ABA) as ideal functionalities (denoted by $\mathcal{F}_{\mathsf{ABA}}$, see Figure 13) in the universal composability (UC) framework [Can01]. Intuitively, the number of rounds is defined as the maximum number of times a party alternates between sending messages to the functionalities and receiving messages from the functionalities. Coretti et al. also give the first constant-round AMPC protocol in [CGHZ16], which is still the most efficient constant-round AMPC without assuming fully homomorphic encryptions (FHE). They follow the BMR framework [BMR90] to do multiparty garbling of the circuit C via a general non-constant-round AMPC [BKR94] and let all the parties evaluate the garbled circuit. However, the resulting communication complexity of their protocol is at least $O(|C|n^3\kappa)$ even if we apply the best general AMPC protocol in this setting [BJK+25]. In contrast, the best non-constant-round AMPC protocols have achieved linear communication of $O(|C|n)$[1] field elements [GLS24, JLS24, BJK+25] in computing an arithmetic circuit (over a large field). This leads us to the following question:

"Can we construct a constant-round AMPC protocol with optimal resilience that achieves communication complexity of $O(|C|n\kappa)$?"

1.1 Our Contribution

In this work, we answer this question affirmatively and achieve the following result. Our construction achieves malicious security with abort in the $\mathcal{F}_{\mathsf{ACS}}$-hybrid model, where $\mathcal{F}_{\mathsf{ACS}}$ with parameter (M, m) allows the parties to agree on a subset of m values over M possible values, which is essential for AMPC protocols. We refer the readers to Section C for the definition of $\mathcal{F}_{\mathsf{ACS}}$. Our protocol only requires 3 sequential invocations of $\mathcal{F}_{\mathsf{ACS}}$, where each $\mathcal{F}_{\mathsf{ACS}}$ can be realized in constant rounds [CGHZ16] assuming (concurrent) ABA and expected constant time [CFG+23] (we discuss the asynchronous round complexity in more detail in Section 1.2). The first 2 invocations of $\mathcal{F}_{\mathsf{ACS}}$ have parameters $(n, n-t)$ and the last one has parameter $(N = O(n+\kappa), N/4)$, and the instantiations of them only requires communication of $O(\mathsf{poly}(n, \kappa))$ bits (see C for more details).

[1] We omit the terms that are sublinear in $|C|$ for the communication of related works introduced in this section.

Among them, the first one is used for instantiating the functionality $\mathcal{F}_{\mathsf{Setup}}$ (see Figure 3) for the setup of our protocol. The other two are used for realizing our main protocol in the $\{\mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{ACS}}\}$-hybrid model.

In particular, we first give our theorem for SIMD (Single Instruction Multiple Data) circuits with can compute the same circuit multiple times with different inputs.

Theorem 1. *Assuming random oracles, in the $\mathcal{F}_{\mathsf{ACS}}$-hybrid model, there exists a computationally secure constant-round asynchronous MPC protocol for SIMD circuits (that contain $\Omega(n+\kappa)^2$ copies of the same circuit) that achieves malicious security with abort against $t = (n-1)/3$ corrupted parties with communication of $O(|C|n\kappa + \mathsf{poly}(n,\kappa))$ plus 3 invocations of $\mathcal{F}_{\mathsf{ACS}}$ with parameters $(n, n-t)$, $(n, n-t)$, and $(N = O(n+\kappa), N/4)$, where $|C|$ is the total circuit size, and κ is the secure parameter.*

Combining with the techniques of network routing [GPS21, GPS22, GLOS25], our protocol can be upgraded to handle general circuits.

Theorem 2. *Assuming random oracles, in the $\mathcal{F}_{\mathsf{ACS}}$-hybrid model, there exists a computationally secure constant-round asynchronous MPC protocol that achieves malicious security with abort against $t = (n-1)/3$ corrupted parties with communication of $O(|C|n\kappa + D(n+\kappa)^2 n\kappa + \mathsf{poly}(n,\kappa))$ plus 3 invocations of $\mathcal{F}_{\mathsf{ACS}}$ with parameters $(n, n-t)$, $(n, n-t)$, and $(N = O(n+\kappa), N/4)$, where $|C|$ is the circuit size, D is the circuit depth, and κ is the secure parameter.*

Compared to the constant-round AMPC protocol in [CGHZ16], our protocol improves the communication complexity by a factor of $O(n^2)$ at the cost of giving up the guaranteed output delivery and assuming random oracles. It is worth noticing that the corruption threshold $t = (n-1)/3$ is still optimal even for malicious security with abort: When $n = 3t$, we may separate the $2t$ honest parties into two sets, each of size t. Then the t corrupted parties can use different inputs to interact with two different sets of t honest parties separately. When the corrupted parties follow the protocol in these two different executions and the network delay between any pair of honest parties in different sets is sufficiently large, the two sets of honest parties must terminate the execution and get their output before receiving any message from honest parties in the other set. This clearly breaks the security since the two sets of honest parties would end up with different outputs.

To achieve our result, we adapt the recent constant-round MPC [GLOS25] in the synchronous setting that achieves $O(|C|\kappa)$ in both communication and garbled circuit size to the asynchronous setting. The main technical difficulty is to adapt the MPC-in-the-head framework used in [GLOS25] to the asynchronous network. We refer the readers to Section 2 for an overview of our techniques.

1.2 Related Works

Constant-Round Synchronous MPCs. In the synchronous setting, the study of constant-round MPC protocols among n parties is initialized by Beaver, Micali, and Rogaway [BMR90]. They present the BMR framework of multiparty garbling, which is the most widely used technique to construct constant-round MPCs without fully homomorphic encryption [Gen09]. A rich line of works follow the BMR framework to improve the efficiency [DI05, LPSY15, BLO16, BLO17, HSS17, HOSS18a, HOSS18b, YWZ20, BCO+21, BGH+23, GLM+25, GLOS25]. In particular, in the setting of honest majority, the best known result [GLOS25] achieves $O(|C|\kappa)$ communication only assuming random oracles.

General AMPCs. AMPC protocol with optimal resilience $n = 3t+1$ is first constructed in the information-theoretic (IT) setting [BKR94]. Then, a line of works [PCR09, CP23, JLS24, GLS24] improves the protocol in the IT setting to achieve $O(|C|n)$ elements of communication. However, the additive overhead of the protocol [GLS24] is $O(n^{14})$ elements, which is far from practice. Another line of works [HNP05, HNP08, CHL21] construct computational AMPC protocols with linear homomorphic encryption schemes. The achieved communication complexity in [CHL21] is $O(|C|n^2)$ elements. The work [CP15] achieves communication of $O(|C|n)$ elements using a stronger assumption of somewhat homomorphic encryptions. Recently, a work [BJK+25] achieves a communication cost of $O(|C|n)$ elements only assuming random oracles.

On Asynchronous Round Complexity. All the asynchronous protocols mentioned in the last paragraph follow the gate-by-gate paradigm to evaluate the circuit, so their round complexities are all linear to the depth of the circuit. For general MPC protocols in the asynchronous setting, strict constant-round protocols do not exist if we only assume point-to-point secure asynchronous channels, as the impossibility results from [FL82, DS83] show that ABA protocols can't terminate in a strict constant number of rounds. However, solutions for ABA [CR93, CKS05] and concurrent ABAs (without causally related inputs and outputs) [CFG+23] with an expected constant number of rounds exist. For this reason, the previous work [CGHZ16] defines the round complexity of AMPCs assuming the ideal functionality of ABA. However, as mentioned in [CFG+23], if several ABAs are run concurrently, and their inputs and outputs are related, these ABAs may not terminate in an expected constant time using a single instantiation of expected constant-time ABA. Therefore, we consider constant-round AMPC assuming the functionality of ACS, which is essential for an AMPC protocol, and it can be instantiated in expected constant time [CFG+23]. Constant-round AMPC constructed in the $\mathcal{F}_{\mathsf{ACS}}$-hybrid model implies an expected constant-time AMPC protocol if all the invocations of $\mathcal{F}_{\mathsf{ACS}}$ are sequential. For a more detailed discussion of asynchronous round complexity, we refer the readers to [CGHZ16, CFG+23].

2 Technical Overview

In this section, we give a high-level overview of the main techniques used in this paper. Recall that our goal is to construct an asynchronous MPC protocol with optimal resilience $t = (n-1)/3$ against a fully-malicious adversary in the random oracle (RO) model, and we allow the parties to abort the protocol if malicious behaviors of the adversary are detected.

2.1 Main Approaches on Constant-Round MPC

We first review previous approaches and then present the main difficulties in constructing constant-round asynchronous MPC (AMPC) protocols with linear communication.

The main technique to construct a constant-round MPC in the synchronous setting is the garbled circuit introduced by Yao [Yao86]. In Yao's garbled circuit, each wire w of a circuit is associated with a tuple $(\lambda_w, k_{w,0}, k_{w,1})$ randomly picked by the garbler, where λ_w is the wire mask for the wire value v_w, and $k_{w,0}$ (resp. $k_{w,1}$) is the key corresponds to masked value $v_w \oplus \lambda_w = 0$ (resp. $v_w \oplus \lambda_w = 1$) of this wire. For each gate g with input wires a, b and output wire o in the circuit, the garbler encrypts the output wire label $k_{o,v_o \oplus \lambda_o} \| (v_o \oplus \lambda_o)$ using input wire keys $(k_{a,v_a \oplus \lambda_a}, k_{b,v_b \oplus \lambda_b})$ corresponding to each pair of the masked values of a, b for $v_o = g(v_a, v_b)$. In this way, with input labels $k_{a,v_a \oplus \lambda_a} \| (v_a \oplus \lambda_a), k_{b,v_b \oplus \lambda_b} \| (v_b \oplus \lambda_b)$, the evaluator can decrypt the output label $k_{o,v_o \oplus \lambda_o}$. During the evaluation of the garbled circuit, the evaluator receives the input labels $k_{i,v_i \oplus \lambda_i} \| (v_i \oplus \lambda_i)$ for each input wire i and decrypts the wire label of each wire gate by gate.

In the multiparty setting, the main approach to achieve constant-round MPC is to let the parties express the computation of the garbler as a constant-depth circuit and jointly compute the garbled circuit under a general MPC protocol [BMR90], which is referred to as multiparty garbling. However, doing it naively requires all parties to securely evaluate cryptographic primitives such as PRG using an MPC protocol. To make black-box use of cryptographic primitives, each party chooses his own keys for encryptions, and the labels of each wire contain the keys chosen by everyone [DI05]. In this way, the resulting garbled circuit size is linear in the number of parties. To achieve linear communication in the number of parties n, a direct approach is to use a packed sharing to compute the garbler's computation and reconstruct the garbled circuit to a single evaluator. The single evaluator will compute the output of the garbled circuit for every party. At the end of the protocol, each party receives the output labels from the evaluator. Each party can check the results by verifying whether the labels are consistent with the chosen keys.

However, in the asynchronous setting, the above approach doesn't work. If the single evaluator is corrupted, he may never send the output. Note that in the synchronous setting, parties can just abort if they do not receive the results from the evaluator within a fixed amount of time. In the asynchronous setting, since parties cannot distinguish a corrupted evaluator not sending the output from an honest evaluator whose

message is delayed due to the network latency, they have to wait, and we lose the liveness. To solve this problem, a natural idea is to let everyone play as the evaluator, which leads to a multiplicative overhead of n on the communication. Thus, in order to achieve linear communication, we need the parties to jointly compute a garbled circuit whose size is independent of the number of parties.

2.2 Review of Constant-Size Multiparty Garbled Circuit

In the synchronous setting, the only known constant-size multiparty garbled circuit that can be computed in the RO model is given by [GLOS25], which achieves security with abort in the honest majority setting of $n = 2t+1$. A natural idea is to adapt the construction in [GLOS25] in the asynchronous setting. We first give a review of their approach. Unlike [BMR90, DI05], [GLOS25] compiles a non-constant-round MPC protocol with specific properties into a constant-round variant. Their compiler enables the evaluator to obtain not only the garbled circuits of the local computations in the original protocol but also the input wire labels for the transcripts. For simplicity, we describe their compiler applied to the well-known DN protocol [DN07].

Achieving Constant-Round. The compiler contains two stages. The first stage compiles the DN protocol with a low corruption threshold of $t = (n-1)/3$ to achieve constant round complexity. The second stage uses the party virtualization technique [Bra87] to boost the corruption threshold to $t = (n-1)/2$.

We first introduce the first stage. In the DN protocol, parties first run a constant-round preprocessing phase to prepare random double-sharing pairs of the form $([r]_t, [r]_{2t})$, (where $[\cdot]_t$ denotes the degree-t Shamir sharing[2]) and a constant-round input phase to share each input value x by $[x]_t$. Then, parties evaluate the circuit in a gate-by-gate fashion. Due to the linearity of the Shamir secret sharing scheme, linear gates can be computed locally. For multiplication gates with input sharings $[x]_t, [y]_t$, the parties locally compute $[z]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$. Then, the parties jointly reconstruct z and compute $[x \cdot y]_t = z - [r]_t$. In this way, the evaluation phase only contains reconstructions of degree-$2t$ sharings, and each party's local computation only relies on the secrets of these sharings and the input/preprocessing data. Letting each party garble his local computation to do the round collapsing, the evaluator only needs the wire labels for preprocessing/input data and the secrets of these degree-$2t$ sharings to simulate the execution of DN protocol.

Now we explain how the evaluator knows the label of the secret of each degree-$2t$ sharings reconstructed in the protocol. The main idea is to let the receiver of each degree-$2t$ sharing choose the labels $k_{w,0}, k_{w,1}$ of each wire w whose value is the secret of the sharing. The receiver shares $[k_{w,0}]_{3t}$ and $[k_{w,1} - k_{w,0}]_t$ at the beginning. With $[z]_t$ in hand, the parties can locally compute

$$[k_{w,z}]_{3t} = [k_{w,0}]_{3t} + [z]_{2t} \cdot [k_{w,1} - k_{w,0}]_t. \tag{1}$$

Crucially, when garbling the local circuits that produce $[z]_{2t}$ in the original MPC protocol, parties also garble the associated label computation circuits. This enables the evaluator to simultaneously reconstruct both the secret z and its corresponding wire label $k_{w,z}$. Then, the evaluator can compute the whole process of the DN protocol. We call the resulting protocol the outer protocol.

Following the above approach, the garbled circuit size is the sum of the garbled circuits of all the parties' local computations. The size of garbled circuits of local computations only depends on the computation complexity of the non-constant-round protocol and the output size. The output labels consist exclusively of the shared labels $[k_{w,0}]_{3t}, [k_{w,1}]_{3t}$ for each wire w whose value is the secret z of a degree-$2t$ sharing $[z]_{2t}$ reconstructed in the evaluation phase. Consequently, these shared labels contribute only a constant factor overhead relative to the DN protocol's communication cost. Given that the DN protocol requires $O(|C|n^2)$ computation and communication complexity[3], the resulting garbled circuit size is $O(|C|n^2\lambda)$.

Applying the ramp secret sharing scheme of [CC06] and the network routing technique [GPS21, GPS22], the authors of [GLOS25] construct a non-constant-round MPC which has similar properties to the DN

[2]For simplicity, we assume that the secrets and shares of the Shamir sharings are binary.

[3]Readers may notice that the original DN protocol only has $O(|C|n)$ computation and communication complexity, rather than $O(|C|n^2)$. This difference is because in the above, each reconstruction of $[z]_{2t}$ is done by sending the whole sharing to every party, which is required by the compiler in [GLOS25], whereas in the original DN protocol, each reconstruction is done by letting a single party reconstruct the secret and then distribute the result back.

protocol, achieving constant AND operations and constant communication complexity at the cost of a lower corruption threshold of $t = n/4$. By employing FreeXOR techniques [KS08], local XOR operations can be handled without affecting the garbled circuit size. Then, in this setting, a constant-size multiparty garbled circuit is constructed.

Boosting the Corruption Threshold. Now we explain the ideas of the second stage of [GLOS25] that boosts the corruption threshold via the party virtualization technique [Bra87]. At the beginning of the protocol, the parties agree on a random set of c parties $P_{i,1}, \ldots, P_{i,c}$ (where c is a constant) to emulate every single party V_i in the above outer protocol. The data of V_i is additively shared among $P_{i,1}, \ldots, P_{i,c}$. While garbling V_i's local circuit, the computation is done via a dishonest majority inner MPC protocol running among $P_{i,1}, \ldots, P_{i,c}$ (using the multiparty garbling introduced in Section 2.1). When the number of virtual parties is $N = O(n+\kappa)$, the authors proved by Chernoff bound that the number of groups formed completely by corrupted parties is at most $N/4$ with overwhelming probability. This means that the adversary only has access to the data of $N/4$ virtual parties in the execution of the outer protocol, which guarantees privacy. If we only consider the semi-honest security, the instantiation of the inner protocol only requires the preparation of random OTs, which could be prepared via OT extension [IKNP03, KOS15] from a small number of base OTs. The base OTs can be prepared by running a general honest majority MPC by all the parties in constant rounds. Since each inner protocol runs among only a constant set of parties, it won't lead to a non-constant multiplication overhead on the communication.

To achieve malicious security, the authors apply the MPC-in-the-head technique [IKOS07, IPS08]. The parties commit their inputs before they participate in the inner protocol. After the inner protocols terminate, each party will choose a public set of virtual parties. The members of each virtual party in this set will open their input set for the virtual party and then send all the transcripts for the computation of this virtual party. In this way, it can be guaranteed that at least $3N/4$ virtual parties are computed honestly and are not checked by the corrupted parties, or the malicious behavior of the adversary will be detected with overwhelming probability. In this way, malicious security is achieved.

2.3 Difficulties Caused by the Asynchrony

Now we analyze the difficulties in computing the above constant-size garbled circuit in the asynchronous setting.

Difficulty 1: Generating the Sharings and Commitments. To adapt the idea of [GLOS25] to the asynchronous setting, we need to first handle how the parties share their inputs. In the synchronous setting, every party can wait for all the input sharings. If a share isn't received within the time bound, a party can abort the protocol. While in the asynchronous setting, a corrupted party may send his input sharings to only a part of the parties, and the rest of the parties may wait forever for the shares. A usual solution to this issue is the technique of asynchronous complete secret sharing (ACSS) [BKR94, PCR09, CP23, JLS24, GLS24], which can ensure that a party distributes a valid sharing to all the parties. However, this technique only works on distributing Shamir sharings, but the sharings used to achieve constant-size garbled circuits are those of [CC06].

Another issue is that the receiver of each degree-$2t$ sharing needs to share the wire labels for the secret. However, corrupted receivers may never distribute these labels, and honest parties cannot distinguish such failures from network delays in the asynchronous setting. Since these labels serve as inputs to the inner protocol for garbling virtual parties' local computations, missing shares prevent progress in the inner protocol.

Finally, the MPC-in-the-head framework poses additional challenges. The parties need to commit their input set for the inner protocols of each virtual party before running them. These committed values should be opened if the corresponding virtual party is checked. In the synchronous setting, the commitment opening is straightforward. If the commitments are not opened or not correctly opened, the verifier can abort the protocol. However, in the asynchronous setting, a corrupted party may never open his commitments, and an honest verifier can't distinguish him from an honest party with high latency. A corrupted party might never

open commitments for a virtual party's inputs, particularly if it cheated during that party's computation. This prevents the verification process from terminating reliably, breaking the security guarantees of the synchronous approach.

Difficulty 2: The Inner Protocol of Virtual Parties. We haven't discussed how to garble each virtual party's local computation yet. A natural idea is to use a general AMPC to perform multiparty garbling, as [CGHZ16] did in constructing the first constant-round AMPC.

However, the inner protocol of [GLOS25] requires a dishonest majority, whereas it is known that asynchronous MPC exists only when the corruption threshold is bounded by $t < n/3$. Due to this reason, we can't simply apply a general AMPC to perform the garbling of virtual parties' computations.

In addition, unlike [GLOS25], parties can't expect to get the garbled circuits from all the parties in the asynchronous setting, as the virtual parties completely emulated by corrupted parties may never send their outputs. Again, because of the asynchrony, such malicious behavior can not be distinguished from the network delay.

Next, we will show how we address the above difficulties.

2.4 Sharing and Committing via AVID

To deal with the first difficulty, we make use of the asynchronous verifiable information dispersal (AVID) technique [CT05, ADD+22]. AVID enables a party to disperse a message (which does not preserve the secrecy of the message) to all the parties, and the parties can help a party retrieve the message. Once the dispersing phase terminates, the retrieving process guarantees termination and correctness.

In order to distribute a sharing, the parties first prepare pairwise symmetric keys, and then disperse the encryption of each party's share by the symmetric key between the two parties. In this way, the whole sharing can be distributed to everyone with eventual delivery by the retrieving process of AVID. The only issue is that the sharings may not be valid. This can be checked by letting each party additionally distribute a random sharing and verify a random linear combination of all the sharings. Note that random coins and symmetric keys can be prepared by a general AMPC protocol. Once all the instances of AVID for a party's message finish their dispersing phases, their validity can be checked by all the parties.

For the sharings of wire labels (which are generated by the receivers in [GLOS25]), we change the generation process of them. Note that the only difference between such a sharing and a completely random sharing is that the receiver knows the random secret. To avoid the case that corrupted receivers don't distribute such sharings, we let all the parties prepare random sharings jointly and then reconstruct the secrets to the corresponding receivers.

Now the remaining problem is how to generate commitments. Similarly, this can also be done via AVID. To let a party P commit a vector $\boldsymbol{v}$:

1. All the parties first jointly prepare a random degree-t Shamir sharing and reconstruct the secret seed seed to P. The degree-t sharing can be prepared via ACSS.

2. Then, P expands the seed with the random oracle and masks $\boldsymbol{v}$ with the result, say $\boldsymbol{v} \oplus \mathcal{O}(\mathsf{seed})$. P then computes $\boldsymbol{v} \oplus \mathcal{O}(\mathsf{seed})$ and disperses it via AVID.

3. When the committed value needs to be opened to some party P', all the parties reconstruct the seed to P' and let P' retrieve the dispersed value.

In this way, the committed vector $\boldsymbol{v}$ can't be changed once the AVID dispersing terminates, and the opening of the commitment guarantees eventual termination.

Let's now summarize the sharing and committing processes:

1. At the beginning, the parties prepare pairwise symmetric keys and the random sharing of seeds used for commitments.

2. The parties generate sharings following the approach of [GLOS25], where the shares are encrypted by pairwise symmetric keys and are dispersed via AVID. In particular, the random sharings for the chosen output labels are jointly prepared by all the parties.

3. Each party P_i generates a commitment for all the sharings he generates for a member $P_{j,\alpha}$ of each virtual party V_j. Specifically speaking, P_i expands a seed for $P_{j,\alpha}$ and uses it to mask the set $\mathsf{Sh}_i^{P_{j,\alpha}}$ of all the shares he generated for $P_{j,\alpha}$. Finally, the masked values are dispersed via AVID as a commitment.

4. The parties agree on a common set $\mathsf{CoreSet}$ of at least $2t+1$ parties who finish the dispersal of all their sharings and commitments. For all the sharings generated by them, the parties compute a random linear combination of them to check their validity.

5. The members of virtual parties sample their local randomness and commit the sets of randomness and shares they receive from each party in $\mathsf{CoreSet}$. The set $\mathsf{IS}^{P_{i,j}}$ of a party $P_{i,j}$ is the input set of $P_{i,j}$ to the computation of virtual party V_i. For each set $\mathsf{IS}^{P_{i,j}}$, $P_{i,j}$ generates a commitment of it and disperses it via AVID as in Step 3.

2.5 MPC-in-the-Head: from Synchronous to Asynchronous

As we have pointed out, we can't ensure that enough virtual parties can be computed securely by an AMPC. To address the second difficulty, a natural idea is to adapt the synchronous MPC running among members of each virtual party [GLOS25] with careful modifications.

Let's first point out the properties of a synchronous protocol:

- In a synchronous MPC, parties are able to wait for all the messages in a round.
- After each round of communication, every party knows that all other parties have also finished this round.

A key observation of ours is that, while running a synchronous protocol in asynchronous networks cannot guarantee termination, we can preserve the two crucial synchronous properties. More concretely, for each virtual party V_i, each member $P_{i,j}$ should commit his input to the inner protocol. Only when $P_{i,j}$ receives the commitment (i.e. the dispersal message) of every $P_{i,\alpha}, \alpha \neq j$, he begins to run the inner protocol. During each round of the inner protocol, the parties not only send messages but also send their commitments via AVID. Each party's commitment shows that he has finished sending messages in the current round. Each party waits for all the messages from other members, together with their commitments. A party will participate in the next round of the inner protocol once he has received all the messages and commitments (i.e. the dispersal messages) in the current round. At the end of the final round of the inner protocol, the members commit their outputs via AVID. When a party receives the output commitments from all the members of a virtual party, he marks the virtual party terminated.

Now, we describe how the N virtual parties are formed. When a virtual party is emulated by c members, the expected number of virtual parties that are completely emulated by corrupted parties is only about $(1/3)^c \cdot N$. On the other hand, the expected number of virtual parties formed by all-honest parties is about $(2/3)^c \cdot N$. For these virtual parties, we can expect their inner protocols to terminate. Thus, if we ensure that most of the other virtual parties, namely those consisting of both honest and corrupted parties, either perform correct computation or do not finish the computation, the parties are able to wait for (a bit less than) $(2/3)^c \cdot N$ virtual parties' results. Among them, only a fraction of (a bit more than) $\frac{(1/3)^c}{(2/3)^c} = 2^{-c}$ may not be correctly computed. This ratio can be sufficiently small to reach the security requirement of the outer protocol.

We observe that in the construction of [GLOS25], the evaluator does not require the garbled circuits from all the virtual parties for the evaluation. Instead, it only needs a sufficient number of garbled circuits to reconstruct the wire labels for each secret reconstructed in the underlying protocol (i.e. $[k_{w,z}]_{3t}$ in Equation 1). For simplicity, we still use the compiled DN protocol as an example. Note that the corruption

threshold for the underlying DN protocol is the number of completely corrupted virtual parties, which is close to $3^{-c}N$. Consequently, only about $3^{1-c}N$ garbled circuits are required, while we can wait for at least $(2^c-1)\cdot 3^{-c}N > 3^{1-c}N$ garbled circuits that are not generated completely by corrupted parties when $c \geq 3$. Therefore, it is sufficient for the parties to perform the evaluation with only $(2/3)^c \cdot N$ virtual parties' results.

The final challenge is how to ensure that most of the virtual parties formed by both honest and corrupted parties correctly run their inner protocols if they terminate. Like [GLOS25], we still follow the idea of MPC-in-the-head [IKOS07, IPS08]. A small issue of applying the idea to the asynchronous setting is that each party only waits for a small fraction of all the virtual parties to terminate their inner protocols, and the parties may have different sets of terminated virtual parties. To ensure that enough terminated virtual parties perform their computations correctly, $O(N)$ virtual parties in each party's set should be checked by honest parties. However, we can't let each party choose $O(N)$ virtual parties to verify, or otherwise, the corrupted parties may have access to too many virtual parties' data.

To solve this issue, we let all the parties agree on a set of terminated parties. For each virtual party in the set, the parties agree on a random coin that decides whether the virtual party is checked by all the parties. Each virtual party is checked with a small constant probability. Since the termination of each virtual party is dispersed to all the parties, once an honest party marks a virtual party terminated, it will be marked by all the honest parties eventually. Thus, all the parties will receive the outputs from the same set of virtual parties, and the verification is performed on a small common subset of $O(N)$ virtual parties. This guarantees correctness. On the other hand, corrupted parties still have access to only a small fraction of the virtual parties' data, which guarantees security. We refer the readers to Section 5 for the detailed construction. Here we give an outline of the garbling and verification process:

1. The members of each virtual party run the inner protocol to garble the virtual party's local computation in the non-constant-round protocol. During each round of the inner protocol, each member commits the set of all the messages sent in this round via AVID. After receiving all the messages and commitments in the current round, parties can continue in the next round. Finally, the members generate commitments for their outputs from the inner protocol.

2. The parties agree on a set Ter of virtual parties that terminate their inner protocols, guaranteed that only a small fraction of the virtual parties in Ter are completely formed by corrupted parties. For each virtual party in Ter, the parties agree on a coin to decide whether its computation is checked.

3. For each checked virtual party, its members open the commitments for the inputs to the inner protocol and the transcripts of the inner protocol to all the parties. The commitments are opened by retrieving the commitments via AVID and reconstructing the seeds. The parties then check whether the computation of the virtual party is correctly performed.

4. If the check passes, all the members of virtual parties of Ter open their output of the inner protocols, i.e. the garbled circuits of the virtual parties' local computation of the non-constant-round protocol.

Finally, all the parties obtain the garbled circuits generated by the virtual parties in Ter (if the verification passes). Then, each party can evaluate the circuit as in [GLOS25] and obtain their outputs. We refer the readers to Section 5 for more details.

3 Preliminaries

Let κ denote the secure parameter, and let $\mathbb{F}_q$ denote the finite field with q elements. We use $\boldsymbol{u} * \boldsymbol{v}$ to denote the coordinate-wise multiplication of two vectors $\boldsymbol{u}, \boldsymbol{v}$ of the same length, and we use $\boldsymbol{u} \otimes \boldsymbol{v}$ to denote the tensor product of two vectors, defined by

$$\boldsymbol{u} \otimes \boldsymbol{v} = (u_i \cdot v_j)_{i\in\{1,\ldots,k\},j\in\{1,\ldots,\ell\}} = (u_1v_1, \ldots, u_1v_\ell, \ldots, u_kv_1, \ldots, u_kv_\ell)$$

for $\boldsymbol{u} = (u_1, \ldots, n_k), \boldsymbol{v} = (v_1, \ldots, v_\ell)$.

Security Model. In our work, we follow the security model in [CGHZ16, Coh16]. We use the UC framework introduced by Canetti [Can01] to define the security of our protocols, based on the *real and ideal world paradigm* [Can00]. Informally, we consider a protocol Π to be secure if its execution in the real world can also be done in the ideal world. For more details about the security model, we refer the readers to Section A.

Asynchronous Rounds. We follow the definition of asynchronous rounds under the assumption of the ideal functionality $\mathcal{F}_{\mathsf{ACS}}$ of the agree on a common set (ACS) protocol (see Figure 16 in Section C). The number of rounds is defined to be the maximum number of times an honest party alternates between sending a message to a secure channel and sending a request to fetch the output from a secure channel.

3.1 Asynchronous Subprotocols

In the remaining sections of this paper, we assume $t = (n-1)/3$. We now introduce some asynchronous subprotocols in this section.

Agree on a Common Set. The agreement on a common subset (ACS) primitive allows the parties to agree on a set of inputs satisfying a certain property.

We give the ACS functionality $\mathcal{F}_{\mathsf{ACS}}$ in Section C. Using the techniques from [BCG93, AJM+23, CFG+23], $\mathcal{F}_{\mathsf{ACS}}$ can be achieved in expected constant time. We refer the readers to Section C for more details.

Asynchronous Verifiable Information Dispersal. An asynchronous verifiable information dispersal (AVID) protocol [CT05] enables a party to disperse a message among a group of parties during a dispersal phase, where the message can be retrieved by any other party in a retrieving phase. We provide the functionality $\mathcal{F}_{\mathsf{AVID}}$ of AVID in Section B (see Figure 14). We borrow the result from [ADD+22] for the instantiation of the AVID functionality.

Theorem 3. ([ADD+22]) *There exists a protocol Π_{AVID} that securely realizes $\mathcal{F}_{\mathsf{AVID}}$ with communication of $O(|M| + n\kappa)$ bits in the dispersal phase to disperse a message M and communication of $O(|M| + n\kappa)$ bits for a party to retrieve a dispersed message M. The dispersal phase requires 6 rounds of communication, and the retrieval phase requires 1 round.*

Random Coin. We will also need a subprotocol for preparing a random common coin. For the formal functionality $\mathcal{F}_{\mathsf{Coin}}$ of generating a random coin, we refer the readers to Section B (Figure 15). The functionality can be realized by letting all the parties prepare a random sharing via ACSS [PCR09, CP23, JLS24] and reconstructing the secret via the online error correction (OEC) process [Can96]. We conclude the following theorem.

Theorem 4. *There exists a constant-round protocol Π_{Coin} that securely realizes $\mathcal{F}_{\mathsf{Coin}}$ with communication of $O(\mathsf{poly}(n, \kappa))$ bits.*

Remark 1. *The instantiation of $\mathcal{F}_{\mathsf{Coin}}$ requires an invocation of $\mathcal{F}_{\mathsf{ACS}}$ to agree on a set of parties' inputs that are correctly shared among all the parties. This ACS process can be run within the first $\mathcal{F}_{\mathsf{ACS}}$ invocation to instantiate $\mathcal{F}_{\mathsf{Setup}}$ (the ACSS processes can be run in parallel with the setup phase of our main protocol, see Section 5 for more details). Thus, this doesn't affect our final result.*

3.2 Linear Secret Sharing Schemes

We introduce the Linear Secret Sharing Schemes (LSSS) we used in our construction. We follow the definition of LSSS from [GLOS25]. We give the formal definition of LSSSs in Section D. Intuitively, a (n, t, k, ℓ)-linear secret sharing scheme (LSSS) Σ over $\mathbb{F}_q$ is a secret sharing among n parties, with a sharing algorithm that maps the secret to n shares and a reconstruction algorithm that maps all the shares to the secret. The secret

is in $\mathbb{F}_q^k$ and each share is in $\mathbb{F}_q^\ell$. Each group of t shares in such a sharing is independent of the secret. An LSSS satisfies *linearity*, which means the sharing and reconstructing algorithms are both linear.

For an LSSS Σ over $\mathbb{F}_q$, a vector of m sharings $([s_1], \ldots, [s_m])$ in Σ can naturally be regarded as an LSSS $\Sigma_{\times m}$ over $\mathbb{F}_{q^m}$. $\Sigma_{\times m}$ is called an m-fold interleaved secret sharing scheme of Σ [CCXY18].

The LSSSs we used are natural extensions of the LSSSs from [CC06, GLOS25]. Our non-constant-round protocol is built upon (N, T, k, ℓ)-LSSSs $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$ over $\mathbb{F}_2$, denoted by $[\cdot], [\cdot]^{(2)}, [\cdot]^{(3)}$ respectively. In the rest of this paper, the notations $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$ are used to denote these specific schemes. It holds that the tensor product of two Σ-sharings is a $\Sigma^{(2)}$-sharing, i.e. $[\boldsymbol{s}^{(1)}] \otimes [\boldsymbol{s}^{(2)}] = [\boldsymbol{s}^{(1)} * \boldsymbol{s}^{(2)}]^{(2)}$. Similarly, the tensor product of a $\Sigma^{(2)}$-sharing and a Σ-sharing is a $\Sigma^{(3)}$-sharing. We take $T = N/20$, $q = 2^{20}$, $k = N/20$, and $\ell = 20$ in our protocol, the resulting $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$ are all $(N, T, k, 20)$-LSSSs over $\mathbb{F}_2$. In addition, the secret of each $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharing can be reconstructed from any $N/5$ shares, and there exists efficient algorithms to decide whether a set of shares is from a valid $\Sigma/\Sigma^{(2)}/\Sigma^{(3)}$ sharing and reconstruct the secret (as observed in [GLOS25]). We give more details about the LSSSs we used in Section D.

4 The Non-Constant-Round protocol

In this section, we give the non-constant-round protocol Π_0, which is a special synchronous MPC protocol that runs among n clients $C_1, \ldots, C_n$ and N servers $S_1, \ldots, S_N$. Our main AMPC construction is given by letting the parties compute the garbled circuit compiled from Π_0. Π_0 is a natural extension of the non-constant-round protocol of [GLOS25]. For simplicity, we give the construction of Π_0 for single instruction multiple data (SIMD) circuits that contain $\Theta((n+\kappa)^2)$ copies of the same circuit. The protocol Π_0 is secure against a semi-honest adversary that corrupts up to $t = (n-1)/3$ clients and $T = N/20$ servers. The construction follows the gate-by-gate fashion based on the secret sharing scheme Σ and the associated sharing $\Sigma^{(2)}, \Sigma^{(3)}$ introduced in Section 3.2 and Section D. We assume that the number of copies of a single circuit contained in the SIMD circuit to compute is a multiple of k^2 times, where k is the secret size of Σ.

The basic idea of the construction is to batch each k gates of the same type together, where the batched gates are in the same position of different single-data circuits of the SIMD circuit. Gates of the same batch are evaluated together. The clients first share their inputs using Σ-sharings for each batch of input gates. During the evaluation, the goal of the servers is to use the Σ-sharings for batched input wires of each group of gates to jointly compute the Σ-sharing for the batched output wires of them. After the whole circuit is evaluated, the servers hold Σ-sharings for batches of output wires. They only need to send each sharings to the client who needs the corresponding output and let him reconstruct his output.

Due to the linearity of Σ, batched addition gates can be evaluated locally. For batched multiplication gates, the parties can locally compute a $\Sigma^{(2)}$-sharing of the output wire values from Σ-sharings of input wire values, and then the parties need a degree reduction process to convert it to a Σ-sharing.

Subprotocols. We follow the degree reduction process of [GLOS25]. We need the following subprotocols to help us perform the degree reduction process:

- A protocol $\Pi_{\mathsf{RandShare}}$ that enables all the servers jointly prepare random $\Sigma^{(2)}$-sharings.
- A protocol $\Pi_{\mathsf{Transpose}}$ that takes $\Sigma^{(2)}$-sharings $[\boldsymbol{x}_1]^{(2)}, \ldots, [\boldsymbol{x}_k]^{(2)}$ as input. Let $\boldsymbol{x}_i = (x_{i,1}, \ldots, x_{i,k})$ and $\boldsymbol{x}_i^* = (x_{1,i}, \ldots, x_{k,i})$, $\Pi_{\mathsf{Transpose}}$ outputs $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$.

With the above subprotocols, we follow [GLOS25] to build the protocol Π_{Multi} that takes Σ-sharings $[\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_k]$ and $[\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_k]$ as inputs. The outputs of Π_{Multi} are $[\boldsymbol{z}_1], \ldots, [\boldsymbol{z}_k]$ with $[\boldsymbol{z}_j] = [\boldsymbol{x}_j * \boldsymbol{y}_j]$ for $j = 1, \ldots, k$. The protocol Π_{Multi} can evaluate k^2 multiplication gates together with communication of $O(N^2)$ bits. We refer the readers to Section E for more details.

Protocol Description. Now we present the protocol Π_0 for SIMD circuits as follows.

Protocol Π_0

Preprocessing Phase

Let C be the SIMD circuit to compute. Let G_A be the number of multiplication gates of C.

1. The servers run $\Pi_{\mathsf{RandShare}}$ to prepare $2G_AN\ell/k^2$ random $\Sigma^{(2)}$-sharings.
2. The servers do the following $2G_A/k^2$ times in parallel:
   (a) Each server S_j generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}_j]^{(2)}$ and distributes it to all the servers.
   (b) The servers group each $N\ell$ random $\Sigma^{(2)}$-sharings obtained from $\Pi_{\mathsf{RandShare}}$ together as $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{N\ell}]^{(2)}$.

   Each group of sharings prepared in this step is associated with an instance of $\Pi_{\mathsf{Transpose}}$.
3. For each batch of k output wires attached to a client C_i, C_i generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}]^{(2)}$ and distributes it to all the servers.
4. The servers agree on a public Σ-sharing $[\mathbf{1}]$ where $\mathbf{1} = (1, 1, \ldots, 1)$.

Input Phase

For each batch of k input wires attached to client C_i, C_i generates a random Σ-sharing $[\boldsymbol{x}]$ of the input vector $\boldsymbol{x}$ on the batch of wires.

Evaluation Phase

1. All parties evaluate the circuit gate by gate:
   (a) **Addition Gates:** For each batch of k addition gates with input sharings $[\boldsymbol{x}], [\boldsymbol{y}]$, all the servers locally compute $[\boldsymbol{x} + \boldsymbol{y}] = [\boldsymbol{x}] + [\boldsymbol{y}]$.
   (b) **Multiplication Gates:** For each k groups of multiplication gates with input sharings $([\boldsymbol{x}_1], [\boldsymbol{y}_1]), \ldots, ([\boldsymbol{x}_k], [\boldsymbol{y}_k])$, all the servers run Π_{Multi} with input sharings $[\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_k]$ and $[\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_k]$.
2. After evaluating all the layers of the circuit and collecting the input sharings for the output layer, for each input sharing $[\boldsymbol{y}]$ for an output gate attached to each client C_i, the servers locally compute $[\boldsymbol{y} + \boldsymbol{r}]^{(2)} = [\mathbf{1}] \otimes [\boldsymbol{y}] + [\boldsymbol{r}]^{(2)}$ with the corresponding $[\boldsymbol{r}]^{(2)}$ and send it to C_i. Then C_i reconstructs $\boldsymbol{y} + \boldsymbol{r}$ and computes $\boldsymbol{y} = \boldsymbol{y} + \boldsymbol{r} - \boldsymbol{r}$ to get his output.

Figure 1: The protocol Π_0.

Note that the evaluation phase of Π_0 only contains local computations and reconstructions of $\Sigma^{(2)}$-sharings. This property is important when we compile it to our main protocol. The communication complexity of Π_0 is $O(|C| + \mathsf{poly}(n, N, \kappa))$, and the servers together need $O(|C|)$ AND operations.

Using the techniques of network routing [GPS21, GPS22], the above protocol can be extended to compute a general circuit. The communication complexity is $O(|C| + DN^2\kappa + \mathsf{poly}(n, N, \kappa))$, and the servers together need $O(|C| + DN^2)$ AND operations. The evaluation phase still consists of local computations and reconstructions of $\Sigma^{(2)}$-sharings. We refer the readers to [GLOS25] for more details.

5 The Main Protocol

In the main protocol, we consider the case that there are n parties, and up to t of them are corrupted, with $n = 3t + 1$. We consider malicious security with abort in the asynchronous setting.

5.1 Parameter Choices for Virtual Servers

In the main protocol, n parties will act as the n clients in Π_0, where each P_i will emulate the client C_i. For the N servers, we use the party virtualization technique [Bra87]. We let 3 parties emulate a virtual server V_j corresponding to the server S_j in Π_0. For the remaining of this section, we consider Π_0 to be performed among $P_1, \ldots, P_n$ (as clients $C_1, \ldots, C_n$) and $V_1, \ldots, V_N$ (as servers $S_1, \ldots, S_N$). Between the parties emulating the same virtual server, the parties run an inner dishonest-majority synchronous MPC protocol Π_{in} to garble the local computation of the server's computation in Π_0.

Taking N random groups of 3 parties to emulate the N virtual servers, the probability that a virtual server is completely honest is larger than $(2/3)^3 = 8/27$. We let $X_i = 1$ if all the parties emulating the i-th virtual server are completely honest and $X_i = 0$ otherwise. Since these N groups of servers are chosen independently, $X_1, \ldots, X_N$ are independent random variables. Let $X = X_1 + \cdots + X_N$, we have $\mu = \mathbb{E}(X) > 8N/27$. By Chernoff bound (see Section F for the lemma of Chernoff bound) it holds that:

$$P(X \leq N/4) \leq P(X \leq (1 - 5/32)\mu) \leq e^{-\frac{25\mu}{2048}} = e^{-\Omega(N)}.$$

Taking the union bound of at most $\binom{n}{t} < 2^n$ possible choices of the set of corrupted parties, the probability that less than $N/4$ virtual servers are honest is no more than $e^{-\Omega(N-n)}$. Let $N = n + \Theta(\kappa) = \Theta(n + \kappa)$, with overwhelming probability, each party can wait for the outputs of Π_{in} from $N/4$ virtual servers. Similarly, the probability that a virtual server is completely corrupted is smaller than $(1/3)^3 = 1/27$, and with overwhelming probability, there will not be more than $N/26$ completely corrupted virtual servers.

From the above analysis, the parties can wait for a common set $N/4$ virtual servers to terminate their computation (which can be determined by the ACS functionality). The resulting set Ter is determined via an invocation of $\mathcal{F}_{\mathsf{ACS}}$. For each of these virtual servers, the parties agree on a random coin that outputs 1 with a probability of 1/320 and outputs 0 with a probability of 319/320. If the random coin outputs 1, the dispersal inputs and transcripts of the corresponding virtual server's inner protocol are retrieved by all the parties. The parties then check whether the virtual server performs the computation correctly. By Chernoff Bound, the probability that more than $N/312$ virtual servers are checked is negligible. Besides, if more than $N/264$ virtual servers formed by both honest and corrupted parties don't perform their computation correctly, the probability that each of them is checked is 1/320, so the probability that none of them is checked is at most

$$(1 - 1/320)^{\frac{N}{264}} = e^{-\Omega(N)},$$

which is negligible.

Another issue we haven't discussed comes from the preprocessing for inner protocols. To let the members of a virtual server perform multiplications securely under a dishonest majority, we need all the parties to prepare random OTs (ROTs) before the inner protocol. For these values from ROTs, the adversary may launch a selective failure attack as noted in [IPS08]. For example, say two parties (P_1, P_2) invoke an instance of ROT where P_1 is corrupted and P_2 is honest. P_1 receives $r_0, r_1 \in \{0,1\}^\kappa$ and P_2 receives $b \in \{0,1\}, r_b$ from the ROT. Then, if $b = 0$, P_1 may commit $r_0, r_1' \neq r_1$ as his output from the ROT, and the multiplication performed with this instance of ROT can still be computed correctly. In this case, the parties would not be able to detect the malicious behavior, and the adversary would learn that $b = 0$, which breaks the security of the multiplication protocol. However, the adversary has at most 1/2 probability to carry out such an attack on each ROT without being caught. If the adversary performs such an attack on over $N/220$ virtual servers in Ter, the probability of catching the adversary when checking any of them is at least $(1/2)\cdot(1/320) = 1/640$. Thus, the total probability of not catching the attack during the verification is at most

$$(1 - 1/640)^{\frac{N}{220}} = e^{-\Omega(N)},$$

which is still negligible.

As analyzed above, with overwhelming probability, at most $N/26 + N/312 + N/264 + N/220 = N/20$ virtual parties in Ter are dishonest, i.e., their data is revealed to the adversary or they do not perform their computation honestly. This matches the corruption threshold $T = N/20$ in Π_0.

5.2 The Inner Protocol for Multiparty Garbling

Now we present the inner protocol Π_{in} that garbles each virtual server's local computation Π_0. We present the formal definition of ROT instances, and we give subprotocols for opening an additive sharing (Π_{Open}) and computing the multiplications of two additively shared values (Π_{Mult}) in Section G. We follow the approach of [LPSY15] to do the multiparty garbling, which supports freeXOR [KS08] in the random oracle model. For simplicity, we assume that the message lengths of the random oracles are compatible with the equations in the protocol, for example, the equation $\mathcal{O}(s) \oplus M$ with $M \in \{0,1\}^m$ implies $\mathcal{O}(s) \in \{0,1\}^m$.

Protocol Π_{in}

The protocol runs between 3 parties $P_{j,1}, P_{j,2}, P_{j,3}$, who agree on a circuit $\mathtt{Circ}^{V_j}$ to be garbled and a circuit C to be computed in the MPC protocol. The circuit $\mathtt{Circ}^{V_j}$ is the circuit of local computation of V_j during the evaluation phase of Π_0, which consists of AND, XOR, and output gates.

Inputs: The parties' inputs consist of the following:

1. $2G_A N(\ell+1)/k^2 + W_I/k + W_O/k$ additive sharings. The secrets of these sharings are V_j's shares of the sharings prepared in the preprocessing phase and the input phase of Π_0, i.e.
   - $2G_A/k^2$ groups of sharings $[\boldsymbol{r}_1]^{(2)}, \ldots, [\boldsymbol{r}_N]^{(2)}$ which are generated in Step 3.(a) of Π_0.
   - $2G_A/k^2$ groups of sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{N\ell}]^{(2)}$ which are grouped in Step 3.(b) in Π_0.
   - A sharing $[\boldsymbol{r}]^{(2)}$ for each batch of k output wires.
   - W_I/k input sharings generated in the input phase of Π_0.

   Here, G_A, W_I, W_O are the numbers of AND gates, input wires, and output wires of C respectively. In addition, the parties input the sharing $\langle \boldsymbol{r}_j \rangle$ for each group.
2. For each wire of $\mathtt{Circ}^{V_j}$ except output wires of XOR and output gates, each party $P_{j,i}$ inputs a wire label $k_{w,0}^{P_{j,i}} \in \mathbb{F}_2^{\kappa-1}$ and his share $\lambda_w^{P_{j,i}}$ of the additive sharing $\langle \lambda_w \rangle$ of wire mask $\lambda_w \in \{0,1\}$.
3. Each party inputs a global $(\kappa-1)$-bit string $\Delta^{P_{j,i}}$ as the sum (over $\mathbb{F}_2$) of his two wire labels of each wire.
4. For each output gate of $\mathtt{Circ}^{V_j}$ (with index k), each party $P_{j,i}$ input two $3\ell\kappa$-bit strings $Y_{\mathsf{k},0}^{P_{j,i}}, Y_{\mathsf{k},1}^{P_{j,i}}$.
5. Each pair of parties input $\kappa-1$ instances of ROTs.
6. For each reconstruction of $\Sigma^{(2)}$-sharing in the evaluation phase of Π_0 whose receiver is V_j, each party $P_{j,\beta}$ inputs 2κ k-bit strings $\boldsymbol{r}_{\mathbf{0},\beta}^{(1)}, \ldots, \boldsymbol{r}_{\mathbf{0},\beta}^{(\kappa)}$ and $\boldsymbol{r}_{\mathbf{1},\beta}^{(1)}, \ldots, \boldsymbol{r}_{\mathbf{1},\beta}^{(\kappa)}$.

The protocol proceeds as follows:

1. **OT Extension.** Each pair of parties invokes the extension in [KOS15] to get $13 \cdot G_A^{V_j} + 6\ell W_O^{V_j}$ instances of ROTs with message length $\kappa-1$. Here $G_A^{V_j}$ is the number of AND gates in $\mathtt{Circ}^{V_j}$ and $W_O^{V_j}$ is the number of output wires in $\mathtt{Circ}^{V_j}$.
2. **Handling XOR gates.** For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, each party $P_{j,i}$ computes
$$k_{o,0}^{P_{j,i}} \| \lambda_o^{P_{j,i}} = (k_{a,0}^{P_{j,i}} \| \lambda_a^{P_{j,i}}) \oplus (k_{b,0}^{P_{j,i}} \| \lambda_b^{P_{j,i}}).$$
   This computation is performed gate by gate.
3. **Computing 1-Labels.** For each wire w in $\mathtt{Circ}^{V_j}$ that is not an output wire of an output gate. Each party $P_{j,i}$ computes $k_{w,1}^{P_{j,i}} = k_{w,0}^{P_{j,i}} \oplus \Delta^{P_{j,i}}$.
4. **Handling AND Gates.** For each AND gate g in $\mathtt{Circ}^{V_j}$ with input wires a, b and output wire o, the parties hold $\langle k_{o,0}^{P_{j,\alpha}} \rangle$ where $P_{j,\alpha}$'s share is $k_{o,0}^{P_{j,\alpha}}$ and all other parties have all-0 shares. Similarly, they hold $\langle k_{o,1}^{P_{j,\alpha}} \rangle$, and they also hold $\langle \lambda_a \rangle, \langle \lambda_b \rangle, \langle \lambda_o \rangle$. Then:
   (a) Each pair of parties take 1 instances of ROT. Then the parties run Π_{Mult} to compute $\langle \lambda_a \cdot \lambda_b \rangle$. Then the parties locally compute $\langle \chi_1 \rangle, \langle \chi_2 \rangle, \langle \chi_3 \rangle, \langle \chi_4 \rangle$, where
$$\chi_1 = ((0 \oplus \lambda_a) \wedge (0 \oplus \lambda_b)) \oplus \lambda_o, \qquad \chi_2 = ((0 \oplus \lambda_a) \wedge (1 \oplus \lambda_b)) \oplus \lambda_o,$$
$$\chi_3 = ((1 \oplus \lambda_a) \wedge (0 \oplus \lambda_b)) \oplus \lambda_o, \qquad \chi_4 = ((1 \oplus \lambda_a) \wedge (1 \oplus \lambda_b)) \oplus \lambda_o.$$
   (b) Each pair of parties takes 12 instances of ROTs. Then the parties run Π_{Mult} to compute $\langle \chi_i \cdot (k_{o,1}^{P_{j,\alpha}} - k_{o,0}^{P_{j,\alpha}}) \rangle$ for $i = 1,2,3,4$ and $\alpha = 1,2,3$. Then the parties locally compute $\langle k_{o,\chi_i}^{P_{j,\alpha}} \rangle = \langle k_{o,0}^{P_{j,\alpha}} \oplus \chi_i \cdot (k_{o,1}^{P_{j,\alpha}} - k_{o,0}^{P_{j,\alpha}}) \rangle$ for $i = 1,2,3,4$ and $\alpha = 1,2,3$.
   (c) Each party $P_{j,i}$ calls the random oracle $\mathcal{O}$ with input $k_{a,i_0}^{P_{j,i}} \| i_0 \| k_{b,i_1}^{P_{j,i}} \| i_1 \| i \| j \| \alpha \| g$ for each $(i_0, i_1) = (0,0), (0,1), (1,0), (1,1)$ and $\alpha = 1,2,3$ and then receives the output. The output can be regarded as additively shared among $P_{j,1}, P_{j,2}, P_{j,3}$, where all the parties except $P_{j,i}$ have all-0 shares.

(d) The parties locally compute $\langle A_{g,2i_0+i_1}^{P_{j,\alpha}} \rangle$ for each $(i_0, i_1) = (0,0), (0,1), (1,0), (1,1)$ and $\alpha = 1,2,3$, where

$$A_{g,2i_0+i_1}^{P_{j,\alpha}} = \left(\bigoplus_{i=1}^{3} \left(\mathcal{O}(k_{a,i_0}^{P_{j,i}} \| i_0 \| k_{b,i_1}^{P_{j,i}} \| i_1 \| i \| j \| \alpha \| g) \right) \right) \oplus (k_{o,\chi_{2i_0+i_1}}^{P_{j,\alpha}} \| \chi_{2i_0+i_1}),$$

Let $\boldsymbol{A}_{g,1}^{V_j} = (A_{g,1}^{P_{j,1}}, A_{g,1}^{P_{j,2}}, A_{g,1}^{P_{j,3}})$ and similar for $\boldsymbol{A}_{g,2}^{V_j}, \boldsymbol{A}_{g,3}^{V_j}, \boldsymbol{A}_{g,4}^{V_j}$. The parties then get $\langle \boldsymbol{A}_{g,1}^{V_j} \rangle, \langle \boldsymbol{A}_{g,2}^{V_j} \rangle, \langle \boldsymbol{A}_{g,3}^{V_j} \rangle, \langle \boldsymbol{A}_{g,4}^{V_j} \rangle$.

5. **Handling Output Gates.** For each output gate with index k in $\mathtt{Circ}^{V_j}$ with input wire w, the parties hold $\langle Y_{\mathsf{k},0}^{P_{j,\alpha}} \rangle$ where $P_{j,\alpha}$'s share is $Y_{\mathsf{k},0}^{P_{j,\alpha}}$ and all other parties have all-0 shares. Similarly, they hold $\langle Y_{\mathsf{k},1}^{P_{j,\alpha}} \rangle$, and they also hold $\langle \lambda_w \rangle$. Then:
   (a) Each pair of parties takes 6ℓ instances of ROTs. Then the parties run Π_{Mult} to compute $\langle (i_2 \oplus \lambda_w) \cdot (Y_{\mathsf{k},1}^{P_{j,\alpha}} - Y_{\mathsf{k},0}^{P_{j,\alpha}}) \rangle$ for $i_2 = 0,1$ and $\alpha = 1,2,3$. Then the parties locally compute $\langle Y_{\mathsf{k},i_2 \oplus \lambda_w}^{P_{j,\alpha}} \rangle = \langle Y_{\mathsf{k},0}^{P_{j,\alpha}} \oplus (i_2 \oplus \lambda_w) \cdot (Y_{\mathsf{k},1}^{P_{j,\alpha}} - Y_{\mathsf{k},0}^{P_{j,\alpha}}) \rangle$ for $i_2 = 0,1$ and $\alpha = 1,2,3$.
   (b) Each party $P_{j,i}$ calls the random oracle $\mathcal{O}$ with input $k_{w,i_2}^{P_{j,\alpha}} \| i_2 \| i \| j \| \alpha \| w$ for each $i_2 = 0,1$ and $\alpha = 1,2,3$ and then receives the output. The output can be regarded as additively shared among $P_{j,1}, P_{j,2}, P_{j,3}$, where all the parties except $P_{j,i}$ have all-0 shares.
   (c) The parties locally compute $\langle \mathsf{ct}_{w,i_2}^{P_{j,\alpha}} \rangle$ for each $i_2 = 0,1$ and $\alpha = 1,2,3$, where

$$\mathsf{ct}_{w,i_2}^{P_{j,\alpha}} = \left(\bigoplus_{i=1}^{3} \left(\mathcal{O}(k_{w,i_2}^{P_{j,i}} \| i_2 \| i \| j \| \alpha \| w) \right) \right) \oplus Y_{\mathsf{k},i_2 \oplus \lambda_w}^{P_{j,\alpha}},$$

   Let $\mathbf{ct}_{w,i_2}^{V_j} = (\mathsf{ct}_{w,i_2}^{P_{j,1}}, \mathsf{ct}_{w,i_2}^{P_{j,2}}, \mathsf{ct}_{w,i_2}^{P_{j,3}})$ for each $i_2 = 0,1$. The parties then get $\langle \mathbf{ct}_{w,0}^{V_j} \rangle, \langle \mathbf{ct}_{w,1}^{V_j} \rangle$.
6. **Masking Input Wire Values.** For each input wire w of $\mathtt{Circ}^{V_i}$, if the wire value x_w doesn't come from a reconstruction of a $\Sigma^{(2)}$-sharing, the servers (holding $\langle x_w \rangle$) compute $\langle x_w \oplus \lambda_w \rangle$ and then run $\Pi_{\mathsf{Open}}(\langle x_w \oplus \lambda_w \rangle)$.
7. **Outputting Output Masks.** For each input wire w of an output gate in $\mathtt{Circ}^{V_j}$, the parties output their shares of $\langle \lambda_w \rangle$.
8. **Encrypting Input Labels.** For each reconstruction of $\Sigma^{(2)}$-sharing in the evaluation phase of Π_0 whose receiver is V_j, we assume it's the i-th reconstruction, and the sharing to be reconstructed is $[\boldsymbol{s}_i]^{(2)}$. Suppose the η-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_η in $\mathtt{Circ}^{V_j}$:
   (a) Each party $P_{j,\beta}$ queries the random oracle $\mathcal{O}$ with inputs $\boldsymbol{r}_{0,\eta,\beta} \| 0 \| i \| \beta \| \eta \| j_\eta$ and $\boldsymbol{r}_{1,\eta,\beta} \| 1 \| i \| \beta \| \eta \| j_\eta$, where $\boldsymbol{r}_{b,\eta,\beta} = (r_{b,\eta,\beta}^{(1)}, \ldots, r_{b,\eta,\beta}^{(\kappa)})$ for each $b = 0,1$ with each $\boldsymbol{r}_{\boldsymbol{b},\beta}^{(\alpha)} = (r_{b,1,\beta}^{(\alpha)}, \ldots, r_{b,k,\beta}^{(\alpha)})$. Then $P_{j,\beta}$ receives $\mathcal{O}(\boldsymbol{r}_{0,\eta,\beta} \| 0 \| i \| \beta \| \eta \| j_\eta)$ and $\mathcal{O}(\boldsymbol{r}_{1,\eta,\beta} \| 1 \| i \| \beta \| \eta \| j_\eta)$.
   (b) Each party $P_{j,\beta}$ locally computes

$$\mathsf{ct}_{j_\eta,0}^{(i,\beta)} = \mathcal{O}(\boldsymbol{r}_{0,\eta,\beta} \| 0 \| i \| \beta \| \eta \| j_\eta) \oplus \left(k_{w_{j_\eta},\lambda_{w_{j_\eta}}}^{P_{j,\beta}} \| \lambda_{w_{j_\eta}} \right)$$

   and

$$\mathsf{ct}_{j_\eta,1}^{(i,\beta)} = \mathcal{O}(\boldsymbol{r}_{1,\eta,\beta} \| 1 \| i \| \beta \| \eta \| j_\eta) \oplus \left(k_{w_{j_\eta},1 \oplus \lambda_{w_{j_\eta}}}^{P_{j,\beta}} \| (1 \oplus \lambda_{w_{j_\eta}}) \right).$$

   Then $P_{j,\beta}$ outputs the ciphertexts $\mathsf{ct}_{j_\eta,0}^{(i,\beta)}, \mathsf{ct}_{j_\eta,1}^{(i,\beta)}$.
9. **Outputting Input Labels.** For each input wire w of $\mathtt{Circ}^{V_j}$, if the wire value x_w doesn't come from a reconstruction of a $\Sigma^{(2)}$-sharing in the evaluation phase of Π_0:
   (a) The parties $P_{j,1}, P_{j,2}, P_{j,3}$ output $x_w \oplus \lambda_w$.
   (b) Each party $P_{j,i}$ outputs $k_{w,x_w \oplus \lambda_w}^{P_{j,i}}$.
10. **Outputting Garbled Circuits.** Each party $P_{j,\beta}$ outputs his shares of $\langle \boldsymbol{A}_{g,1}^{V_j} \rangle, \langle \boldsymbol{A}_{g,2}^{V_j} \rangle, \langle \boldsymbol{A}_{g,3}^{V_j} \rangle, \langle \boldsymbol{A}_{g,4}^{V_j} \rangle$ for each AND gate and $\langle \mathbf{ct}_{w,0}^{V_j} \rangle, \langle \mathbf{ct}_{w,1}^{V_j} \rangle$ for each output wire w.

Figure 2: The inner protocol.

5.3 Main Protocol Description

Now we are ready to introduce our construction of the main protocol Π, which runs the Setup Phase, the Sharing Phase, the Local Computation Phase, the Garbling Phase, the Verification Phase, and the Evaluation Phase in order.

Let $\mathcal{N}$ be the matrix

$$\mathcal{N} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & b_1 & \cdots & b_{2t} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & b_1^t & \cdots & b_{2t}^t \end{pmatrix},$$

where $1, b_1, \ldots, b_{2t}$ are $2t+1$ different elements in $\mathbb{F}_{2^a}$ for $a = \lfloor \log n \rfloor + 1$.

When we say a party aborts the protocol, he also sends abort to all the parties. When a party receives abort from any other party, he also aborts the protocol. Each message of the protocol is associated with a message index mid.

Setup Phase. In the Setup phase, each pair of parties agrees on a pair-wise symmetric key. All the parties determine the virtual servers and prepare seeds for commitments. Each pair of members of a virtual server also prepares base ROTs. We give the functionality $\mathcal{F}_{\mathsf{Setup}}$ of the Setup phase below in Figure 3.

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

The trusted party interacts with all the parties $P_1, \ldots, P_n$ and the adversary Sim. The parties agree on public parameters $\alpha_0, \ldots, \alpha_n \in \mathbb{F}_{2^\kappa}$

1. **Preparing Symmetric Keys.** For each pair of parties (P_i, P_j), the trusted party samples a random κ-bit string as the pair-wise key $k_{i,j} = k_{j,i}$ between P_i and P_j. For each honest party P_i, the trusted party sends $k_{i,j} = k_{j,i}$ as a request-based delayed output to P_i. For each corrupted party P_i and honest party P_j, the trusted party sends $k_{i,j} = k_{j,i}$ to Sim. Similarly, for each group of three parties (P_i, P_j, P_α), the trusted party samples a random κ-bit string as the pair-wise key $k_{i,j,\alpha}$ among them. Sim then sends it to the three parties (to Sim if among them there is a corrupted party) as request-based delayed outputs.
2. **Determining the Virtual Servers.** Let $N = n + \Theta(\kappa)$, the trusted party randomly samples a 3-party set $\{P_{j,1}, P_{j,2}, P_{j,3}\}$ for $j = 1, \ldots, N$. Then, the trusted party sends all these sets to Sim and to honest parties as request-based delayed outputs. The servers $P_{j,1}, P_{j,2}, P_{j,3}$ will run the inner protocol Π_{in} to emulate the virtual server V_j in Π_0.
3. **Preparing the Seeds for Commitments.**
   (a) For each party P_i, each $j = 1, \ldots, N$, and $\alpha = 1, 2, 3$:
   The trusted party randomly samples a κ-bit string as $\mathsf{seed}_i^{P_{j,\alpha}}$ (which is also regarded as an element in $\mathbb{F}_{2^\kappa}$). The trusted party then receives an element in $\mathbb{F}_{2^\kappa}$ as $q_i^{P_{j,\alpha}}(\alpha_u)$ for each corrupted party P_u from Sim. The trusted party then samples a random degree-t polynomial $q_i^{P_{j,\alpha}}(\cdot)$ based on the points received from Sim and $q_i^{P_{j,\alpha}}(\alpha_0) = \mathsf{seed}_i^{P_{j,\alpha}}$. The trusted party then sends $q_i^{P_{j,\alpha}}(\alpha_v)$ to each honest party P_v as a request-based delayed output. If P_i is honest, the trusted party sends $\mathsf{seed}_i^{P_{j,\alpha}}$ to P_i as a request-based delayed output. Otherwise, the trusted party sends $\mathsf{seed}_i^{P_{j,\alpha}}$ to Sim.
   (b) For each $i = 1, \ldots, N$ and $j = 1, 2, 3$:
   The trusted party randomly samples 3 κ-bit strings as $\mathsf{seed}^{P_{i,j}}, \mathsf{seed}^{P'_{i,j}}, \mathsf{seed}_0^{P_{i,j}}$ (which are also regarded as an elements in $\mathbb{F}_{2^\kappa}$). The trusted party then receives 3 elements in $\mathbb{F}_{2^\kappa}$ as $q^{P_{i,j}}(\alpha_u), q^{P'_{i,j}}(\alpha_u), q_0^{P_{i,j}}(\alpha_u)$ for each corrupted party P_u from Sim. The trusted party then samples 3 random degree-t polynomials $q^{P_{i,j}}(\cdot), q^{P'_{i,j}}(\cdot), q_0^{P_{i,j}}(\cdot)$ based on the points received from Sim and $q^{P_{i,j}}(\alpha_0) = \mathsf{seed}^{P_{i,j}}, q^{P'_{i,j}}(\alpha_0) = \mathsf{seed}^{P'_{i,j}}, q_0^{P_{i,j}}(\alpha_0) = \mathsf{seed}_0^{P_{i,j}}$. The trusted party then sends $q^{P_{i,j}}(\alpha_v), q^{P'_{i,j}}(\alpha_v), q_0^{P_{i,j}}(\alpha_v)$ to each honest party P_v as a request-based delayed output. If $P_{i,j}$ is

honest, the trusted party sends $\mathsf{seed}^{P_{i,j}}, \mathsf{seed}^{P'_{i,j}}, \mathsf{seed}_0^{P_{i,j}}$ to $P_{i,j}$ as a request-based delayed output. Otherwise, the trusted party sends $\mathsf{seed}^{P_{i,j}}, \mathsf{seed}^{P'_{i,j}}, \mathsf{seed}_0^{P_{i,j}}$ to Sim.

4. **Prepare ROT instances.** For each virtual server V_j, each pair of parties $(P_{j,\alpha}, P_{j,\beta})$, $P_{j,\beta}$ prepare $\kappa - 1$ instances of random OTs, i.e. the trusted party randomly samples $r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)} \in \{0,1\}^{\kappa-1}$ and $b^{(\alpha,\beta)} \in \{0,1\}$. If $P_{j,\alpha}$ is honest, the trusted party sends $r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)}$ to $P_{j,\alpha}$ as a request-based delayed output. Otherwise, the trusted party sends $r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)}$ to Sim. If $P_{j,\beta}$ is honest, the trusted party sends $b^{(\alpha,\beta)}, r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)}$ to $P_{j,\beta}$ as a request-based delayed output. Otherwise, the trusted party sends $b^{(\alpha,\beta)}, r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)}$ to Sim.

Figure 3: The functionality of the setup phase of Π.

As we can see, the circuit that computes $\mathcal{F}_{\mathsf{Setup}}$ is independent of the main circuit C to be computed. Thus, we can use a general AMPC (whose round complexity does not depend on the number of parties, like [CGHZ16]) to realize $\mathcal{F}_{\mathsf{Setup}}$ with communication of $O(\mathsf{poly}(n, \kappa))$ bits.

Subprotocols to Prepare Virtual Servers' Inputs. We now present subprotocols to prepare the virtual servers' inputs to the inner protocols. The presented protocols are run after invoking $\mathcal{F}_{\mathsf{Setup}}$.

We first show how to compute inputs of $2G_A N(\ell+1)/k^2 + W_O/k$ additive sharings for shares of random $\Sigma^{(2)}$-sharings, i.e.

- $2G_A/k^2$ groups of sharings $[\boldsymbol{r}_1]^{(2)}, \ldots, [\boldsymbol{r}_N]^{(2)}$.
- $2G_A/k^2$ groups of sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{N\ell}]^{(2)}$.
- A sharing $[\boldsymbol{r}]^{(2)}$ for each batch of k output wires.

For each $\boldsymbol{r}_j$, $P_{j,1}, P_{j,2}, P_{j,3}$ additionally need to input $\langle \boldsymbol{r}_j \rangle$. Thus, we prepare 3 random $\Sigma^{(2)}$-sharings $[\boldsymbol{r}_{j,1}]^{(2)}, [\boldsymbol{r}_{j,2}]^{(2)}, [\boldsymbol{r}_{j,3}]^{(2)}$ and reconstruct each secret $\boldsymbol{r}_{j,i}$ to $P_{j,i}$. Then, we set $[\boldsymbol{r}_j]^{(2)} = [\boldsymbol{r}_{j,1}]^{(2)} + [\boldsymbol{r}_{j,2}]^{(2)} + [\boldsymbol{r}_{j,3}]^{(2)}$ and $\langle \boldsymbol{r}_j \rangle = (\boldsymbol{r}_{j,1}, \boldsymbol{r}_{j,2}, \boldsymbol{r}_{j,3})$. For the random sharings prepared for output wires, we similarly split each of them into 3 random sharings.

In this way, we need to prepare $2G_A N(\ell+3)/k^2 + 3W_O/k$ random $\Sigma^{(2)}$-sharings, where each share is shared among 3 members of a virtual server. We prepare them by a share-and-compute process. In the first phase, the parties generate random sharings by themselves. In the second phase, the members of virtual servers to local computations for randomness extraction. In particular, to prepare each $a(t+1)$ random $\Sigma^{(2)}$-sharings (where $a = \lfloor \log n \rfloor + 1$), the parties first run the following protocol $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$.

Protocol $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$

For each party P_i:

1. P_i generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{v}'_i]_a^{(2)}$ and a random $\Sigma^{(2)}$-sharing $[\boldsymbol{o}_i]_a^{(2)}$ where $\boldsymbol{o}_i$ is an all-zero vector. For each share $\boldsymbol{v}_i^{V_j}$ (of $[\boldsymbol{v}'_i]_a^{(2)}$) and $\boldsymbol{o}_i^{V_j}$ (of $[\boldsymbol{o}_i]_a^{(2)}$) for virtual server V_j, P_i generates $\langle \boldsymbol{v}_i^{V_j} \rangle = (\boldsymbol{v}_{i,1}^{V_j}, \boldsymbol{v}_{i,2}^{V_j}, \boldsymbol{v}_{i,3}^{V_j}), \langle \boldsymbol{o}_i^{V_j} \rangle = (\boldsymbol{o}_{i,1}^{V_j}, \boldsymbol{o}_{i,2}^{V_j}, \boldsymbol{o}_{i,3}^{V_j})$ among $P_{j,1}, P_{j,2}, P_{j,3}$.
2. For each $j = 1, \ldots, N$, we assume $P_{j,\alpha} = P_{j_\alpha}$ for $\alpha = 1, 2, 3$. P_i sends $\boldsymbol{v}_{i,1}^{V_j} \oplus \mathcal{O}(k_{i,j_1} \| \mathsf{mid}), \boldsymbol{v}_{i,2}^{V_j} \oplus \mathcal{O}(k_{i,j_2} \| \mathsf{mid}), \boldsymbol{v}_{i,3}^{V_j} \oplus \mathcal{O}(k_{i,j_3} \| \mathsf{mid})$ to $\mathcal{F}_{\mathsf{AVID}}$ (the three messages $\boldsymbol{v}_{i,1}^{V_j}, \boldsymbol{v}_{i,2}^{V_j}, \boldsymbol{v}_{i,3}^{V_j}$ are encrypted by $k_{i,j_1}, k_{i,j_2}, k_{i,j_3}$ respectively). Here $\mathcal{O}$ is a random oracle with output length ℓ^2. Similarly for $\boldsymbol{o}_{i,1}^{V_j} \oplus \mathcal{O}(k_{i,j_1} \| \mathsf{mid}), \boldsymbol{o}_{i,2}^{V_j} \oplus \mathcal{O}(k_{i,j_2} \| \mathsf{mid}), \boldsymbol{o}_{i,3}^{V_j} \oplus \mathcal{O}(k_{i,j_3} \| \mathsf{mid})$, where $\boldsymbol{v}_{i,\beta}^{V_j} \oplus \mathcal{O}(k_{i,j_\beta} \| \mathsf{mid})$ and $\boldsymbol{o}_{i,\beta}^{V_j} \oplus \mathcal{O}(k_{i,j_\beta} \| \mathsf{mid})$ are sent to the same instance of $\mathcal{F}_{\mathsf{AVID}}$.
3. Upon receiving $\mathsf{Dispersed}$ from the instance of $\mathcal{F}_{\mathsf{AVID}}$ corresponding to key $k_{i,j}$, the parties send $(\mathsf{Retrieve}, P_j)$ to $\mathcal{F}_{\mathsf{AVID}}$. For simplicity, we simply say P_i distributes $[\boldsymbol{v}_i]_a^{(2)} = [\boldsymbol{v}'_i]_a^{(2)} + [\boldsymbol{o}_i]_a^{(2)}$ to all the virtual servers via AVID for the above process, and similar for the rest of the protocol.

Figure 4: The sharing process for preparing random $\Sigma^{(2)}$-sharings.

The protocol $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ is invoked in the sharing phase of the main protocol. At the end of the sharing phase, the parties will agree on a common set $\mathsf{CoreSet}$. All the messages of sharings generated by parties in $\mathsf{CoreSet}$ are guaranteed to terminate their corresponding dispersal phase of AVID. After $\mathsf{CoreSet}$ is determined, the parties will run the following protocol $\Pi^{(2)}_{\mathsf{RandShare\text{-}Compute}}$ in the local computation phase.

Protocol $\Pi^{(2)}_{\mathsf{RandShare\text{-}Compute}}$

1. After retrieving all the messages in the last step of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$, the parties locally compute their shares use the pair-wise keys to decrypt the messages. Each group of parties $P_{j,1}, P_{j,2}, P_{j,3}$ then hold additive sharings $\langle \boldsymbol{v}_i^{V_j} \rangle, \langle \boldsymbol{o}_i^{V_j} \rangle$ generated by each party $P_i \in \mathsf{CoreSet}$.
2. Let $\mathsf{CoreSet} = \{P_{i_1}, \ldots, P_{i_{2t+1}}\}$. For each $j = 1, \ldots, N$, the parties $P_{j,1}, P_{j,2}, P_{j,3}$ locally compute
$$\begin{pmatrix} \langle \boldsymbol{w}_1^{V_j} \rangle \\ \langle \boldsymbol{w}_2^{V_j} \rangle \\ \vdots \\ \langle \boldsymbol{w}_{t+1}^{V_j} \rangle \end{pmatrix} = \mathcal{N} \cdot \begin{pmatrix} \langle \boldsymbol{v}_{i_1}^{V_j} \rangle + \langle \boldsymbol{o}_{i_1}^{V_j} \rangle \\ \langle \boldsymbol{v}_{i_2}^{V_j} \rangle + \langle \boldsymbol{o}_{i_2}^{V_j} \rangle \\ \vdots \\ \langle \boldsymbol{v}_{i_{2t+1}}^{V_j} \rangle + \langle \boldsymbol{o}_{i_{2t+1}}^{V_j} \rangle \end{pmatrix},$$
where $\{\boldsymbol{w}_i^{V_j}\}_{j=1}^N$ form a $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{w}_i]^{(2)}_a$ for $i = 1, \ldots, t+1$.
3. For each $j = 1, \ldots, N$, the parties $P_{j,1}, P_{j,2}, P_{j,3}$ locally compute the additive sharing of V_j's shares of $[\boldsymbol{w}_i^{(1)}]^{(2)}, \ldots, [\boldsymbol{w}_i^{(a)}]^{(2)}$ from $\langle \boldsymbol{w}_i^{V_j} \rangle$ for each $i = 1, \ldots, t+1$.

Figure 5: The local computation process for preparing random $\Sigma^{(2)}$-sharings.

For the input Σ-sharings, we can simply let each client generate and share them via AVID. For each wire that is not an output wire for XOR/output gates or an input wire whose value is the secret of a $\Sigma^{(2)}$-sharing to be reconstructed in a virtual server's local circuit, we can let the members generate it by themselves, and similar for the global $(\kappa-1)$-bit strings.

Now the remaining problem is how to let the parties prepare output wire labels together with the random strings for reconstructions of $\Sigma^{(2)}$-sharings, i.e., the strings $Y_{\mathsf{k},0}, Y_{\mathsf{k},1}$ in Π_{in} and $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}, \boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta}$ for $\alpha = 1, \ldots, \kappa$. Recall that we need each evaluator to locally compute the input labels for the secret from the output labels for the shares of each $\Sigma^{(2)}$-sharing.

For each reconstruction of $\Sigma^{(2)}$-sharing in the evaluation of Π_0 and each $\beta = 1, 2, 3$, the parties jointly prepare κ random $\Sigma^{(3)}$-sharings $[\boldsymbol{r}^{(1)}_{\boldsymbol{0},\beta}]^{(3)}, \ldots, [\boldsymbol{r}^{(\kappa)}_{\boldsymbol{0},\beta}]^{(3)}$ and κ random Σ-sharings $[\boldsymbol{r}^{(1)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(1)}_{\boldsymbol{0},\beta}], \ldots, [\boldsymbol{r}^{(\kappa)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\kappa)}_{\boldsymbol{0},\beta}]$. If the receiver of the $\Sigma^{(2)}$-sharing in Π_0 is a virtual server V_j, the parties reconstruct the secrets of the prepared $\Sigma, \Sigma^{(3)}$-sharings to $P_{j,\beta}$. If the receiver is a client C_j, the parties reconstruct the secrets of the sharings for all $\beta = 1, 2, 3$ to the client. For the wires outputting shares of a $\Sigma^{(2)}$-sharing $[\boldsymbol{s}]^{(2)}$ to be reconstructed in Π_0, the output labels of $[\boldsymbol{s}]^{(2)}$ are set to be
$$[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s},\beta}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta}].$$
Thus, for the a-th bit b of $[\boldsymbol{s}]^{(2)}$, the corresponding output wire label is set to be the $a\ell+1$ to $(a+1)\ell$-th bits of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ plus $b \cdot [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ for each $\alpha = 1, \ldots, \kappa$ and $\beta = 1, 2, 3$. In this way, when an evaluator gets the output labels of $[\boldsymbol{s}]^{(2)}$, i.e. the sharings $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s},\beta}]^{(3)}$, he can reconstruct $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s},\beta}$ for $\alpha = 1, \ldots, \kappa$ and $\beta = 1, 2, 3$, which are the input wire labels corresponding to $\boldsymbol{s}$.

To make the above approach work, we require the parties to jointly prepare random Σ and $\Sigma^{(3)}$-sharings. We use $\Pi^{(1)}_{\mathsf{RandShare\text{-}Share}}, \Pi^{(1)}_{\mathsf{RandShare\text{-}Compute}}$ and $\Pi^{(3)}_{\mathsf{RandShare\text{-}Share}}, \Pi^{(3)}_{\mathsf{RandShare\text{-}Compute}}$ to denote the protocols to prepare them. We only need to use $\Sigma, \Sigma_{\times a}$-sharings and $\Sigma^{(3)}, \Sigma^{(3)}_{\times a}$-sharings to replace the $\Sigma^{(2)}, \Sigma^{(2)}_{\times a}$-sharings in $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}, \Pi^{(2)}_{\mathsf{RandShare\text{-}Compute}}$. We omit the details. Assume that there are altogether rec reconstructions of $\Sigma^{(2)}$-sharings $[\boldsymbol{s}_1]^{(2)}, \ldots, [\boldsymbol{s}_{\mathsf{rec}}]^{(2)}$ in the evaluation phase of Π_0, $3\kappa\mathsf{rec}$ pairs of random $\Sigma, \Sigma^{(3)}$-sharings are required.

Remark 2. *For parallel executions of* $\Pi^{(1)}_{\mathsf{RandShare\text{-}Share}}, \Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}, \Pi^{(3)}_{\mathsf{RandShare\text{-}Share}}$*, all the shares sent from a party* P_i *to a party* P_j *can be sent together via a single invocation of* $\mathcal{F}_{\mathsf{AVID}}$*.*

Sharing Phase. Now we are ready to present the sharing phase. The sharing phase is performed after the setup phase terminates. In the Sharing Phase, the parties generate input sharings and random sharings and use AVID to share them. Besides, parties prepare local randomness for the inner protocol. For all these shares and randomness used in the inner protocol, the parties commit them.

Protocol Π_{Share}

Sharing Phase

Let the local circuit of each virtual server V_j in Π_0 be $\mathtt{Circ}^{V_j}$. The parties do the following, where all the messages masked with the same key (i.e., all messages of form $m \oplus \mathcal{O}(k\|\mathsf{mid})$ with the same k) are sent to the same instance of $\mathcal{F}_{\mathsf{AVID}}$ in this phase:

1. **Preparing Random $\Sigma^{(2)}$-Sharings.** Let $a = \lfloor \log n \rfloor + 1$. The parties run $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ $2G_A N(\ell+3)/a(t+1)k^2 + 3W_O/a(t+1)k$ times, where G_A is the number of AND gates in C and W_O is the number of output gates in C. Among them:
   - $2G_A/a(t+1)k^2$ invocations are used to prepare $2G_A/k^2$ groups of $[\boldsymbol{r}_{\eta,i}]^{(2)}$ for $\eta = 1, \ldots, N$ and each $i = 1, 2, 3$. During the executions of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ to prepare $[\boldsymbol{r}_{\eta,i}]^{(2)}$, for each $P_{j,\beta}$ share $v^{V_j}_{\alpha,\beta}$ of the additive sharing of V_j's share of $[\boldsymbol{v}'_\alpha]^{(2)}$, P_α sends $\boldsymbol{v}^{V_j}_{\alpha,\beta} \oplus \mathcal{O}(k_{\alpha,\eta_i,j_\beta}\|\mathsf{mid})$ to $\mathcal{F}_{\mathsf{AVID}}$ (where we assume $P_{\eta,i} = P_{\eta_i}$ and $P_{j,\beta} = P_{j_\beta}$). Then all the parties send $(\mathsf{Retrieve}, P_{\eta,i})$ and $(\mathsf{Retrieve}, P_{j,\beta})$ to $\mathcal{F}_{\mathsf{AVID}}$ upon receiving $\mathsf{Dispersed}$ from it. Upon receiving $\boldsymbol{v}^{V_j}_{\alpha,\beta} \oplus \mathcal{O}(k_{\alpha,\eta_i,j_\beta}\|\mathsf{mid})$ and $\boldsymbol{v}^{V_j}_{\alpha,\beta} \oplus \mathcal{O}(k_{\alpha,j_\beta}\|\mathsf{mid})$, $P_{j,\beta}$ decrypts $\boldsymbol{v}^{V_j}_{\alpha,\beta}$ from the two messages and checks whether they are the same. If not, $P_{j,\beta}$ aborts the protocol.
   - $2G_A N\ell/a(t+1)k^2$ instances of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ are used to prepare $2G_A N/k^2$ groups of random sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{N\ell}]^{(2)}$.
   - For each $a(t+1)$ batches of k output wires attached to a client C_i (emulated by party P_i), an instances of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ is used to prepare the random sharings $[\boldsymbol{r}]^{(2)}$ attached to these batches of output wires. During the executions of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ to prepare $[\boldsymbol{r}]^{(2)}$ for output wires attached to P_i, for each share $\boldsymbol{v}^{V_j}_{\alpha,\beta}$ (for $P_{j,\beta}$) of V_j' share of $[\boldsymbol{v}'_\alpha]^{(2)}$, P_α sends $\boldsymbol{v}^{V_j}_{\alpha,\beta} \oplus \mathcal{O}(k_{\alpha,i,j_\beta}\|\mathsf{mid})$ to $\mathcal{F}_{\mathsf{AVID}}$ (where $P_{j,\beta} = P_{j_\beta}$). All the parties send $(\mathsf{Retrieve}, P_i)$ and $(\mathsf{Retrieve}, P_{j,\beta})$ to $\mathcal{F}_{\mathsf{AVID}}$ upon receiving $\mathsf{Dispersed}$ from it. Upon receiving $\boldsymbol{v}^{V_j}_{\alpha,\beta} \oplus \mathcal{O}(k_{\alpha,i,j_\beta}\|\mathsf{mid})$ and $\boldsymbol{v}^{V_j}_{\alpha,\beta} \oplus \mathcal{O}(k_{\alpha,j_\beta}\|\mathsf{mid})$, $P_{j,\beta}$ decrypts $\boldsymbol{v}^{V_j}_{\alpha,\beta}$ from the two messages and checks whether they are the same. If not, $P_{j,\beta}$ aborts the protocol.
2. **Preprocessing for the Verification of Sharings.** Let $\kappa' = N + \kappa$. Each party P_i generates a random $\Sigma_{\times\kappa'}$-sharing $[\boldsymbol{r}^{(i)}_1]_{\kappa'}$, a random $\Sigma^{(2)}_{\times\kappa'}$-sharing $[\boldsymbol{r}^{(i)}_2]^{(2)}_{\kappa'}$, and a random $\Sigma^{(3)}_{\times\kappa'}$-sharing $[\boldsymbol{r}^{(i)}_3]^{(3)}_{\kappa'}$. Each party P_i generates a random $\Sigma_{\times\kappa'}$-sharing $[\boldsymbol{o}^{(i)}_1]_{\kappa'}$, a random $\Sigma^{(2)}_{\times\kappa'}$-sharing $[\boldsymbol{o}^{(i)}_2]^{(2)}_{\kappa'}$, and a random $\Sigma^{(3)}_{\times\kappa'}$-sharing $[\boldsymbol{o}^{(i)}_3]^{(3)}_{\kappa'}$ with all-zero secrets. Then P_i distributes them to the virtual servers via AVID.
3. **Sharing Inputs.** For each batch of k input wires attached to P_i in circuit C with input values $s_1, \ldots, s_k \in \mathbb{F}_2$, P_i randomly generates $[\boldsymbol{s}]$, where $\boldsymbol{s} = (s_1, \ldots, s_k)$. Then P_i distributes the sharing to the virtual servers via AVID.
4. **Preparing for the Garbling of Local Circuits.**
   (a) **Preparing for the Output Labels.** Let $a = \lfloor \log n \rfloor + 1$. The parties run $\Pi^{(1)}_{\mathsf{RandShare\text{-}Share}}$ and $\Pi^{(3)}_{\mathsf{RandShare\text{-}Share}}$ $3\kappa\mathsf{rec}/a(t+1)$ times. Among them, each $\kappa/a(t+1)$ pairs of invocations of $\Pi^{(1)}_{\mathsf{RandShare\text{-}Share}}$ and $\Pi^{(3)}_{\mathsf{RandShare\text{-}Share}}$ are used to prepare $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ $(\alpha = 1, \ldots, \kappa)$ for each $\beta = 1, 2, 3$ associated with a reconstruction of $\Sigma^{(2)}$-sharing in the evaluation phase of Π_0. For the instances of $\mathcal{F}_{\mathsf{AVID}}$ in these invocations:
      - If the receiver of the $\Sigma^{(2)}$-sharing is a client C_i (emulated by P_i), during the executions of $\Pi^{(1)}_{\mathsf{RandShare\text{-}Share}}, \Pi^{(3)}_{\mathsf{RandShare\text{-}Share}}$ attached to this reconstruction, for each share $\boldsymbol{v}^{V_j}_{\alpha,\beta}$ (for $P_{j,\beta}$) of the additive sharing of V_j's share of $[\boldsymbol{v}'_\alpha]$ or $[\boldsymbol{v}'_\alpha]^{(3)}$, P_α sends $\boldsymbol{v}^{V_j}_{\alpha,\beta} \oplus \mathcal{O}(k_{\alpha,i,j_\alpha}\|\mathsf{mid})$ to $\mathcal{F}_{\mathsf{AVID}}$ (where we

assume $P_{j,\beta} = P_{j_\beta}$). Then all the parties send $(\mathsf{Retrieve}, P_i)$ and $(\mathsf{Retrieve}, P_{j,\beta})$ to $\mathcal{F}_{\mathsf{AVID}}$ upon receiving $\mathsf{Dispersed}$ from it. Upon receiving $\boldsymbol{v}_{\alpha,\beta}^{V_j} \oplus \mathcal{O}(k_{\alpha,i,j_\beta} \| \mathsf{mid})$ and $\boldsymbol{v}_{\alpha,\beta}^{V_j} \oplus \mathcal{O}(k_{\alpha,j_\beta} \| \mathsf{mid})$, $P_{j,\beta}$ decrypts $\boldsymbol{v}_{\alpha,\beta}^{V_j}$ from the two messages and checks whether they are the same. If not, $P_{j,\beta}$ aborts the protocol.

- If the receiver of the $\Sigma^{(2)}$-sharing is a virtual server V_η, let $P_{\eta,i} = P_{\eta_i}$. During the executions of $\Pi^{(1)}_{\mathsf{RandShare\text{-}Share}}, \Pi^{(3)}_{\mathsf{RandShare\text{-}Share}}$ attached to this reconstruction, for each share $v_{\alpha,\beta}^{V_j}$ of the additive sharing of V_j's share of $[\boldsymbol{v}'_\alpha]$ or $[\boldsymbol{v}'_\alpha]^{(3)}$, P_α sends $\boldsymbol{v}_{\alpha,\beta}^{V_j} \oplus \mathcal{O}(k_{\alpha,\eta_i,j_\alpha} \| \mathsf{mid})$ to $\mathcal{F}_{\mathsf{AVID}}$ (where we assume $P_{j,\beta} = P_{j_\beta}$). Then all the parties send $(\mathsf{Retrieve}, P_{\eta,i})$ and $(\mathsf{Retrieve}, P_{j,\beta})$ to $\mathcal{F}_{\mathsf{AVID}}$ upon receiving $\mathsf{Dispersed}$ from it. Upon receiving $\boldsymbol{v}_{\alpha,\beta}^{V_j} \oplus \mathcal{O}(k_{\alpha,\eta_i,j_\beta} \| \mathsf{mid})$ and $\boldsymbol{v}_{\alpha,\beta}^{V_j} \oplus \mathcal{O}(k_{\alpha,j_\beta} \| \mathsf{mid})$, $P_{j,\beta}$ decrypts $\boldsymbol{v}_{\alpha,\beta}^{V_j}$ from the two messages and checks whether they are the same. If not, $P_{j,\beta}$ aborts the protocol.

(b) **Generating Local Randomness.** For each virtual server V_j with local circuit $\mathtt{Circ}^{V_j}$, the parties $P_{j,1}, P_{j,2}, P_{j,3}$ do the following:

i. Each party $P_{j,i}$ samples a random $(\kappa-1)$-bit string as $\Delta^{P_{j,i}}$.

ii. For each wire w that is not an output wire of an XOR gate or an output gate, each party $P_{j,i}$ samples a random bit $\lambda_w^{P_{j,i}}$ as his share of $\langle \lambda_w \rangle$ and a random $(\kappa-1)$-bit string as $k_{w,0}^{P_{j,i}}$.

5. **Committing Sharings.** Let the shares of the sharings generated by each party P_i for each $P_{j,\alpha}$ in this phase form a set $\mathsf{Sh}_i^{P_{j,\alpha}}$. Then, P_i sends $\mathsf{Sh}_i^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}_i^{P_{j,\alpha}} \| \mathsf{mid})$ to $\mathcal{F}_{\mathsf{AVID}}$.

6. **Determining the Core Set.** The parties initialize an ACS functionality $\mathcal{F}_{\mathsf{ACS}}$ with parameter $m = n - t, M = n$. If P_i receives $\mathsf{Dispersed}$ from all the invocations of $\mathcal{F}_{\mathsf{AVID}}$ where the dealer is P_j in the previous steps, P_i sends j to $\mathcal{F}_{\mathsf{ACS}}$. Finally, the parties obtain the set $\mathsf{CoreSet}$ of at least $n - t$ parties from $\mathcal{F}_{\mathsf{ACS}}$. For all the instances of $\mathcal{F}_{\mathsf{AVID}}$ whose dealer P_i is in $\mathsf{CoreSet}$, the parties wait for the output $\mathsf{Dispersed}$ from $\mathcal{F}_{\mathsf{AVID}}$ and then send $(\mathsf{Retrieve}, P_j)$ to $\mathcal{F}_{\mathsf{AVID}}$ for each message M encrypted by $k_{i,j}$.

7. **Committing Local Inputs.** Let the set of all the shares whose encryptions are retrieved for $P_{j,i}$ (i.e. the masked shares are sent to $\mathcal{F}_{\mathsf{AVID}}$, and $(\mathsf{Retrieve}, P_{j,i})$ is sent to this instance of $\mathcal{F}_{\mathsf{AVID}}$ by all the parties) generated by parties in $\mathsf{CoreSet}$, all the ROT outputs from $\mathcal{F}_{\mathsf{Setup}}$, and all the local randomness of a party $P_{j,i}$ be $\mathsf{IS}^{P_{j,i}}$. Upon receiving all these the shares generated by parties in $\mathsf{CoreSet}$ from $\mathcal{F}_{\mathsf{AVID}}$, $P_{j,i}$ sends $\mathsf{IS}^{P_{j,i}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,i}} \| \mathsf{mid})$ to $\mathcal{F}_{\mathsf{AVID}}$.

Figure 6: The sharing phase of Π.

Local Computation Phase. In the local computation phase, the members emulating each virtual server locally compute their input to Π_{in}. Once a party $P_{j,i}$ terminates the local computation phase, he obtains all his inputs to Π_{in} for the virtual server V_j. The local computation phase only contains the parties' local computations.

Protocol Π_{Local}

Local Computation Phase

For each virtual server V_j with local circuit $\mathtt{Circ}^{V_j}$ in Π_0, the parties $P_{j,1}, P_{j,2}, P_{j,3}$ do the following:

1. **Computing Random $\Sigma^{(2)}$-Sharings.** For each execution of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ in Step 1 of the Sharing Phase, the parties run the corresponding protocol $\Pi^{(2)}_{\mathsf{RandShare\text{-}Compute}}$ and obtain the following sharings:

   - $2G_A/k^2$ groups of sharings $[\boldsymbol{r}_{\eta,1}]^{(2)}, [\boldsymbol{r}_{\eta,2}]^{(2)}, [\boldsymbol{r}_{\eta,3}]^{(2)}$ for $\eta = 1, \ldots, N$. After the retrieving process of all the instances of $\mathcal{F}_{\mathsf{AVID}}$ terminates, each $P_{j,i}$ can additionally compute $\boldsymbol{r}_{j,i}$, which is a linear combination of $\{\boldsymbol{v}'_\alpha\}_{\alpha=1}^n$ during the execution of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ for preparing $[\boldsymbol{r}_{j,i}]^{(2)}$, where each sharing $[\boldsymbol{v}'_\alpha]^{(2)}$ is retrieved by $P_{j,i}$ from $\mathcal{F}_{\mathsf{AVID}}$ (same below). The parties then set

   $$[\boldsymbol{r}_j]^{(2)} = [\boldsymbol{r}_{j,1}]^{(2)} + [\boldsymbol{r}_{j,2}]^{(2)} + [\boldsymbol{r}_{j,3}]^{(2)}$$

   and $\langle \boldsymbol{r}_j \rangle = (\boldsymbol{r}_{j,1}, \boldsymbol{r}_{j,2}, \boldsymbol{r}_{j,3})$.
   - $2G_A/k^2$ groups of sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{N\ell}]^{(2)}$.

– A random sharing $[\boldsymbol{r}]^{(2)}$ for each batch of k output wires attached to a client, where the client gets the secret after the retrieving process of the instances of AVID terminates.

2. **Computing Output Labels.**

(a) For each execution of $\Pi^{(1)}_{\mathsf{RandShare\text{-}Share}}, \Pi^{(3)}_{\mathsf{RandShare\text{-}Share}}$ in Step 4.(b) of the Sharing Phase, the parties run the corresponding protocols $\Pi^{(1)}_{\mathsf{RandShare\text{-}Compute}}, \Pi^{(3)}_{\mathsf{RandShare\text{-}Compute}}$ and obtain the sharings $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ for $\alpha = 1, \ldots, \kappa$ and $\beta = 1, 2, 3$ associated with each reconstruction of $\Sigma^{(2)}$-sharing in the evaluation phase of Π_0. After the retrieving process of the instances of AVID invoked in Step 4.(b) terminates, secrets are obtained by P_i if the receiver of the sharing is C_i in Π_0, and the secrets are obtained by $P_{j,\beta}$ for each $\beta = 1, 2, 3$ if V_j is the receiver.

(b) For each $i = 1, \ldots, \mathsf{rec}$, let the associated sharings for the i-th reconstruction be $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ for $\alpha = 1, \ldots, \kappa$ and $\beta = 1, 2, 3$. For $a = 1, \ldots, \ell^2$ and $\beta = 1, 2, 3$, the parties locally compute the additive sharings

$$\langle Y^{V_j}_{(i-1)\ell^2+a,0,\beta} \rangle = \langle ([\boldsymbol{r}^{(1)}_{\mathbf{0},\beta}]^{(3)})^{V_j}_{[a\ell+1,(a+1)\ell]}, \ldots, ([\boldsymbol{r}^{(\kappa)}_{\mathbf{0},\beta}]^{(3)})^{V_j}_{[a\ell+1,(a+1)\ell]} \rangle$$

$$\begin{aligned}\langle Y^{V_j}_{(i-1)\ell^2+a,1,\beta} \rangle = \langle & ([\boldsymbol{r}^{(1)}_{\mathbf{0},\beta}]^{(3)})^{V_j}_{[a\ell+1,(a+1)\ell]} + ([\boldsymbol{r}^{(1)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(1)}_{\mathbf{0},\beta}])^{V_j}, \\ & \ldots, ([\boldsymbol{r}^{(\kappa)}_{\mathbf{0},\beta}]^{(3)})^{V_j}_{[a\ell+1,(a+1)\ell]} + ([\boldsymbol{r}^{(\kappa)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\kappa)}_{\mathbf{0},\beta}])^{V_j} \rangle,\end{aligned}$$

where $([\boldsymbol{s}])^{V_j}$ denotes V_j's share of $[\boldsymbol{s}]$ and $([\boldsymbol{s}]^{(3)})^{V_j}_{[c_1,c_2]}$ denotes the vector of the $c_1, c_1+1, \ldots, c_2$-th bits of V_j's share of $[\boldsymbol{s}]^{(3)}$ (correspond to the i-th reconstruction).

(c) Each party $P_{j,\beta}$ sets $Y^{P_{j,\beta}}_{\mathsf{k},b} = (\langle Y^{V_j}_{\mathsf{k},b,1} \rangle^{P_{j,\beta}}, \ldots, \langle Y^{V_j}_{\mathsf{k},b,c} \rangle^{P_{j,\beta}})$ for each $\mathsf{k} = 1, \ldots, \ell^2\mathsf{rec}$ and $b = 0, 1$, where $\langle s \rangle^{P_{j,\beta}}$ denotes $P_{j,\beta}$'s share of $\langle s \rangle$.

Figure 7: The local computation phase of Π.

Garbling Phase. In the garbling phase, the parties run the inner protocol Π_{in} of multiparty garbling for each virtual server.

Protocol Π_{Garble}

Garbling Phase

For each virtual server V_j, each member $P_{j,i}$ emulating it waits until he receives Dispersed from all the instances of $\mathcal{F}_{\mathsf{AVID}}$ generated in the last step of the sharing phase whose dealer is another party $P_{j,\alpha}, \alpha \neq i$ emulating V_j and then participates in the inner protocol Π_{in}.

During each round of Π_{in}:

1. **Sending Messages.** Each party $P_{j,i}$ computes and sends all the messages he needs to send in Π_{in} to other parties.
2. **Committing Messages.** Let the set of messages sent by $P_{j,i}$ in this round be $\mathsf{MS}^{P_{j,i}}$, $P_{j,i}$ sends $\mathsf{MS}^{P_{j,i}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,i}} \| \mathsf{mid})$ to $\mathcal{F}_{\mathsf{AVID}}$.
3. **Waiting for Termination.** Each party $P_{j,i}$ waits for all the Dispersed messages from $\mathcal{F}_{\mathsf{AVID}}$ which commit $\mathsf{MS}^{P_{j,\alpha}}, \alpha \neq i$. After all these messages are received, $P_{j,i}$ continues to participate in the next round of Π_{in}.

After terminating the last round of Π_{in}, each party $P_{j,i}$ gets his output set $\mathsf{OS}^{P_{j,i}}$. Then, $P_{j,i}$ sends $\mathsf{OS}^{P_{j,i}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,i}} \| \mathsf{mid})$ to $\mathcal{F}_{\mathsf{AVID}}$.

Figure 8: The garbling phase of Π.

Verification Phase. In the verification phase, the parties verify the validity of the sharings generated in the sharing phase. Besides, the parties agree on a common set of virtual servers that terminate their inner protocols. For each virtual server in this set, the parties agree on a common coin to decide whether this

virtual server is checked. For those checked virtual servers, the parties verify whether the committed inputs and transcripts for the virtual servers' computations are consistent.

Protocol Π_{Ver}

Verification Phase

1. **Verification of the Sharings.** The virtual servers run the verification process below to verify all the $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharings generated by parties in $\mathsf{CoreSet}$ in the Sharing Phase. We denote the Σ-sharings to be checked by $[\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_{k_1}]$, the $\Sigma^{(2)}$-sharings by $[\boldsymbol{y}_1]^{(2)}, \ldots, [\boldsymbol{y}_{k_2}]^{(2)}$, and the $\Sigma^{(3)}$-sharings by $[\boldsymbol{z}_1]^{(3)}, \ldots, [\boldsymbol{z}_{k_3}]^{(3)}$, where each virtual server V_j's share is shared by an additive sharing among $P_{j,1}, P_{j,2}, P_{j,3}$. The parties view the sharings to be checked as $\Sigma_{\times\kappa'}, \Sigma^{(2)}_{\times\kappa'}, \Sigma^{(3)}_{\times\kappa'}$-sharings. This step can be run by each party P_i after he receives $\mathsf{CoreSet}$.
   (a) The parties invoke $\mathcal{F}_{\mathsf{Coin}}$ to get $s \in \mathbb{F}_{2^{\kappa'}}$. If abort is received, they abort the protocol. Then the parties expand s to a vector $(s_1, \ldots, s_{k_1}, s_1^{(2)}, \ldots, s_{k_2}^{(2)}, s_1^{(3)}, \ldots, s_{k_3}^{(3)}) \in \mathbb{F}_{2^{\kappa'}}^{k_1+k_2+k_3}$ via random oracle.
   (b) The parties $P_{\alpha,1}, P_{\alpha,2}, P_{\alpha,3}$ of each virtual server V_α locally compute an additive sharing of V_α's share of $[\tau_1]_{\kappa'} = \sum_{j=1}^{k_1} s_j \cdot [\boldsymbol{x}_j]_{\kappa'} + \sum_{P_i \in \mathsf{CoreSet}} [\boldsymbol{r}^{(i)}]_{\kappa'}$ by computing
   $$\sum_{j=1}^{k_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{P_i \in \mathsf{CoreSet}} \langle \boldsymbol{r}_1^{(i,V_\alpha)} \rangle,$$
   where $\boldsymbol{x}_j^{V_\alpha}, \boldsymbol{r}_1^{(i,V_\alpha)}$ are V_α's shares of $[\boldsymbol{x}_j]_{\kappa'}, [\boldsymbol{r}_1^{(i)}]_{\kappa'}$ respectively. Similarly, $P_{\alpha,1}, P_{\alpha,2}, P_{\alpha,3}$ compute an additive sharing of V_α's share of $[\tau_2]^{(2)}_{\kappa'} = \sum_{j=1}^{k_2} s_j^{(2)} \cdot [\boldsymbol{y}_j]^{(2)}_{\kappa'} + \sum_{P_i \in \mathsf{CoreSet}} [\boldsymbol{r}_2^{(i)}]^{(2)}_{\kappa'}$ by computing
   $$\sum_{j=1}^{k_2} s_j^{(2)} \cdot \langle \boldsymbol{y}_j^{V_\alpha} \rangle + \sum_{P_i \in \mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_\alpha)} \rangle$$
   where $\boldsymbol{y}_j^{V_\alpha}, \boldsymbol{r}_2^{(i,V_\alpha)}$ are V_α's shares of $[\boldsymbol{y}_j]^{(2)}_{\kappa'}, [\boldsymbol{r}_2^{(i)}]^{(2)}_{\kappa'}$ respectively, and compute an additive sharing of V_α's share of $[\tau_3]^{(3)}_{\kappa'} = \sum_{j=1}^{k_3} s_j^{(3)} \cdot [\boldsymbol{z}_j]^{(3)}_{\kappa'} + \sum_{P_i \in \mathsf{CoreSet}} [\boldsymbol{r}_3^{(i)}]^{(3)}_{\kappa'}$ by computing
   $$\sum_{j=1}^{k_3} s_j^{(3)} \cdot \langle \boldsymbol{z}_j^{V_\alpha} \rangle + \sum_{P_i \in \mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_\alpha)} \rangle$$
   where $\boldsymbol{z}_j^{V_\alpha}, \boldsymbol{r}_3^{(i,V_\alpha)}$ are V_α's shares of $[\boldsymbol{z}_j]^{(3)}_{\kappa'}, [\boldsymbol{r}_3^{(i)}]^{(3)}_{\kappa'}$ respectively.
   (c) For each $\alpha = 1, \ldots, N$ and $i = 1, 2, 3$, let $P_{\alpha,i}$'s shares of the additive sharings of V_α's shares of $[\tau_1]_{\kappa'}, [\tau_2]^{(2)}_{\kappa'}, [\tau_3]^{(3)}_{\kappa'}$ form a set $\mathsf{VS}^{P_{\alpha,i}}$.

   Then, the parties do the same verification on the $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharings with all-zero secrets to get additive sharings of V_α's shares of $[\tau_1']_{\kappa'}, [\tau_2']^{(2)}_{\kappa'}, [\tau_3']^{(3)}_{\kappa'}$. Each $P_{\alpha,i}$ then adds his shares of them into $\mathsf{VS}^{P_{\alpha,i}}$ and sends $\mathsf{VS}^{P_{\alpha,i}} \oplus \mathcal{O}(\mathsf{seed}^{P'_{\alpha,i}} \| \mathsf{mid})$ to $\mathcal{F}_{\mathsf{AVID}}$.
2. **Verification of Local Computation.**
   (a) The parties initialize an ACS functionality $\mathcal{F}_{\mathsf{ACS}}$ with parameters $m = N/4, M = N$. For each party P_i, upon receiving the $\mathsf{Dispersed}$ messages for all the instances of $\mathcal{F}_{\mathsf{AVID}}$ whose dealer is $P_{j,\alpha}, \alpha = 1, 2, 3$ called in during the execution of Π_{in} for V_j and Step 1 of the verification phase, P_i sends j to $\mathcal{F}_{\mathsf{ACS}}$. Finally, the parties receive Ter from $\mathcal{F}_{\mathsf{ACS}}$.
   (b) For each virtual server $V_j \in \mathsf{Ter}$, the parties invoke $\mathcal{F}_{\mathsf{Coin}}$ to decide whether V_j is required to be checked. The probability that each $V_j \in \mathsf{Ter}$ needs to be checked is $1/320$.
   (c) For each virtual server $V_j \in \mathsf{Ter}$, all the parties send $(\mathsf{Retrieve}, P_i)$ to all the instances of $\mathcal{F}_{\mathsf{AVID}}$ that commit $\mathsf{VS}^{P_{j,\alpha}}, \alpha = 1, 2, 3$ for all $i = 1, \ldots, n$. Then, the parties run the OEC process to let everyone reconstruct $\mathsf{seed}^{P'_{j,\alpha}}$. When the parties retrieve $\mathsf{VS}^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}^{P'_{j,\alpha}} \| \mathsf{mid})$ and get $\mathsf{seed}^{V'_{j,\alpha}}$, they can locally compute $\mathsf{VS}^{P_{j,\alpha}}$ and get V_j's shares of $[\tau]_{\times\kappa'}, [\tau_2]^{(2)}_{\times\kappa'}, [\tau_3]^{(3)}_{\times\kappa'}$ and $[\tau']_{\times\kappa'}, [\tau_2']^{(2)}_{\times\kappa'}, [\tau_3']^{(3)}_{\times\kappa'}$. Then

the parties check whether these shares are valid shares for $\Sigma_{\times\kappa'}, \Sigma^{(2)}_{\times\kappa'}, \Sigma^{(3)}_{\times\kappa'}$-sharings and $\Sigma_{\times\kappa'}, \Sigma^{(2)}_{\times\kappa'}, \Sigma^{(3)}_{\times\kappa'}$-sharings of all-zero secrets.

(d) For each checked virtual server V_j, all the parties sends $(\mathsf{Retrieve}, P_i)$ to all the instances of $\mathcal{F}_{\mathsf{AVID}}$ that commit the following:

- $\mathsf{Sh}_\alpha^{P_{j,\beta}}$ for each $\alpha = 1, \ldots, n$ and $\beta = 1, 2, 3$ committed in Step 5 of the Sharing Phase.
- $\mathsf{IS}^{P_{j,\beta}}$ for each $\beta = 1, 2, 3$ committed in Step 7 of the Sharing Phase.
- $\mathsf{MS}^{P_{j,\beta}}$ for each $\beta = 1, 2, 3$ committed in each round of the inner protocol.
- $\mathsf{OS}^{P_{j,\beta}}$ for each $\beta = 1, 2, 3$ committed at the end the inner protocol.

Then, the parties run the OEC process to let everyone reconstruct $\mathsf{seed}_\alpha^{P_{j,\beta}}, \mathsf{seed}^{P_{j,\beta}}, \mathsf{seed}_0^{P_{j,\beta}}$ for each $\alpha = 1, \ldots, n$ and $\beta = 1, 2, 3$. The parties then locally compute the committed values upon receiving the encryption of them from $\mathcal{F}_{\mathsf{AVID}}$.

(e) For each checked virtual party V_j, the parties do the following:

- The parties check whether $P_{j,1}, P_{j,2}, P_{j,3}$ perform the inner protocol Π_{in} of V_j (from inputs computed from $\mathsf{IS}^{P_{j,1}}, \mathsf{IS}^{P_{j,2}}, \mathsf{IS}^{P_{j,3}}$) and Step 1 of the verification phase (from inputs computed from $\mathsf{Sh}_i^{P_{j,1}}, \mathsf{Sh}_i^{P_{j,2}}, \mathsf{Sh}_i^{P_{j,3}}$ for $P_i \in \mathsf{CoreSet}$) correctly, where the transcripts of Π_{in} for V_j is obtained from $\mathsf{MS}^{P_{j,\beta}}$ for each $\beta = 1, 2, 3$ committed in each round of the inner protocol.
- The parties check whether the additive sharings of V_j's shares committed by the generator (from $\mathsf{Sh}_i^{P_{j,\alpha}}$ for each generator P_i and $\alpha = 1, 2, 3$) match the values committed by the members of V_j (from $\mathsf{IS}^{P_{j,\alpha}}$).
- The parties check whether for each pair of servers $(P_{j,\alpha}, P_{j,\beta})$, the committed ROT outputs from $\mathcal{F}_{\mathsf{Setup}}$ is valid, i.e. $r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)}$ committed by $P_{j,\beta}$ is among $r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)}$ committed by $P_{j,\alpha}$.

If the checks fail, the parties abort the protocol. If all the checks pass, the parties send $(\mathsf{Retrieve}, P_i)$ to the instances of $\mathcal{F}_{\mathsf{AVID}}$ that commit $\mathsf{OS}^{P_{j,\alpha}}$ for all $i = 1, \ldots, n$, $V_j \in \mathsf{Ter}$, and $\alpha = 1, 2, 3$. Then the parties run the OEC process to reconstruct $\mathsf{seed}_0^{P_{j,\beta}}$ to all the parties for each $V_j \in \mathsf{Ter}$ and $\beta = 1, 2, 3$.

Figure 9: The verification phase of Π.

Evaluation Phase. In the evaluation phase, the parties evaluate each virtual server's garbled circuit and compute their outputs.

Protocol Π_{Eval}

Evaluation Phase

1. **Reconstructing the Garbled Circuit.** After passing the checks in the verification phase, the parties retrieve all the encrypted outputs of inner protocols for each $V_j \in \mathsf{Ter}$. Upon finishing the reconstruction process of $\mathsf{seed}_0^{P_{j,\beta}}$ for $\beta = 1, 2, 3$, the parties obtain the outputs of Π_{in} for all $V_j \in \mathsf{Ter}$. Then, the parties reconstruct $A_{g,1}^{P_{j,\beta}}, A_{g,2}^{P_{j,\beta}}, A_{g,3}^{P_{j,\beta}}, A_{g,4}^{P_{j,\beta}}$ for each AND gate and $\mathsf{ct}_{w,0}^{P_{j,\beta}}, \mathsf{ct}_{w,1}^{P_{j,\beta}}$ for each output wire w of $\mathtt{Circ}^{V_j}$.
2. **Evaluating the Circuit.** Each party P_i evaluates the circuit by computing $k_{w,x_w \oplus \lambda_w}^{P_{j,\beta}}$ for each $j = 1, \ldots, N$ and $\beta = 1, 2, 3$. Finally, each party computes the output values and labels for the $\Sigma^{(2)}$-sharings to be reconstructed in the evaluation phase of Π_0 one by one. More concretely, each party P_i evaluates the circuit by running Π_{Ev}, where the detailed description of Π_{Ev} is presented in Section H. After the evaluation, P_i gets $[\boldsymbol{s}_i]^{(2)}$ together with $\{\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}\}_{\alpha=1}^{\kappa}$ for $\beta = 1, 2, 3$ for each $i = 1, \ldots, \mathsf{rec}$.
3. **Computing Outputs.** For each $[\boldsymbol{s}_i]^{(2)}$ whose receiver is a client C_j (emulated by P_j), P_j checks whether $\{\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}\}_{\alpha=1}^{\kappa}$ matches $\boldsymbol{s}_i$ and $\{(\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}, \boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)})\}_{\alpha=1}^{\kappa}$ for each $\beta = 1, 2, 3$ computed in the local computation phase. If not, P_j aborts the protocol. Otherwise, P_j computes his output locally from these secrets $\boldsymbol{s}_i$.

Figure 10: The evaluation phase of Π.

Protocol Summary. The main protocol is presented in Figure 11.

Protocol Π

1. All the parties invoke $\mathcal{F}_{\mathsf{Setup}}$ and wait for the outputs.
2. All the parties run $\Pi_{\mathsf{Share}}, \Pi_{\mathsf{Local}}, \Pi_{\mathsf{Garble}}, \Pi_{\mathsf{Ver}}, \Pi_{\mathsf{Eval}}$ successively.

Figure 11: The main protocol Π.

Theorem 5. *The protocol Π securely realizes the functionality $\mathcal{F}$ in the $\{\mathcal{F}_{\mathsf{ACS}}, \mathcal{F}_{\mathsf{Setup}}, \mathcal{F}_{\mathsf{AVID}}, \mathcal{F}_{\mathsf{Coin}}\}$-hybrid model.*

We give the proof of this theorem in Section I and give a detailed analysis of the communication and round complexity of the protocol in Section J. The total communication complexity is $O(|C|n\kappa + \mathsf{poly}(n, \kappa))$. Then we conclude our main theorem.

Theorem 1. *Assuming random oracles, in the $\mathcal{F}_{\mathsf{ACS}}$-hybrid model, there exists a computationally secure constant-round asynchronous MPC protocol for SIMD circuits (that contain $\Omega(n + \kappa)^2$ copies of the same circuit) that achieves malicious security with abort against $t = (n-1)/3$ corrupted parties with communication of $O(|C|n\kappa + \mathsf{poly}(n, \kappa))$ plus 3 invocations of $\mathcal{F}_{\mathsf{ACS}}$ with parameters $(n, n-t)$, $(n, n-t)$, and $(N = O(n + \kappa), N/4)$, where $|C|$ is the total circuit size, and κ is the secure parameter.*

Applying the protocol to the non-constant-round protocol for general circuits in [GLOS25], we have the following result.

Theorem 2. *Assuming random oracles, in the $\mathcal{F}_{\mathsf{ACS}}$-hybrid model, there exists a computationally secure constant-round asynchronous MPC protocol that achieves malicious security with abort against $t = (n-1)/3$ corrupted parties with communication of $O(|C|n\kappa + D(n + \kappa)^2 n\kappa + \mathsf{poly}(n, \kappa))$ plus 3 invocations of $\mathcal{F}_{\mathsf{ACS}}$ with parameters $(n, n-t)$, $(n, n-t)$, and $(N = O(n+\kappa), N/4)$, where $|C|$ is the circuit size, D is the circuit depth, and κ is the secure parameter.*

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.

Y. Song was also supported in part by the Shanghai Qi Zhi Institute Innovation Program SQZ202313.

References

[ADD+22] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Brief announcement: Asynchronous verifiable information dispersal with near-optimal communication. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing*, pages 418–420. ACM, 2022.

[AJM+23] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. *Distributed Comput.*, 36(3):219–252, 2023.

[BCG93] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 52–61. ACM, 1993.

[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 Franccois-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 12698 of *Lecture Notes in Computer Science*, pages 33–63. Springer, 2021.

[Bea90] Donald Beaver. Multiparty protocols tolerating half faulty processors. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 560–572, New York, NY, 1990. Springer New York.

[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*, 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*, pages 1–10. ACM, 1988.

[BJK+25] Akhil Bandarupalli, Xiaoyu Ji, Aniket Kate, Chen-Da Liu-Zhang, and Yifan Song. Computationally efficient asynchronous MPC with linear communication and low additive overhead. *CRYPTO 2025*, 2025.

[BKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In James H. Anderson, David Peleg, and Elizabeth Borowsky, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 183–192. ACM, 1994.

[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*, 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*, 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*, pages 503–513. ACM, 1990.

[Bra87] Gabriel Bracha. An o(log n) expected rounds randomized byzantine generals protocol. *J. ACM*, 34(4):910–920, 1987.

[Can96] Ran Canetti. Studies in secure multiparty computation and applications. *Scientific Council of The Weizmann Institute of Science*, 1996.

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

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001*, pages 136–145. IEEE Computer Society, 2001.

[Can20] Ran Canetti. Universally composable security. *J. ACM*, 67(5):28:1–28:94, 2020.

[CC06] Hao Chen and Ronald Cramer. Algebraic geometric secret sharing schemes and secure multiparty computations over small fields. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference*, volume 4117 of *Lecture Notes in Computer Science*, pages 521–536. Springer, 2006.

[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*, pages 11–19. ACM, 1988.

[CCXY18] Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized complexity of information-theoretically secure MPC revisited. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference*, volume 10993 of *Lecture Notes in Computer Science*, pages 395–426. Springer, 2018.

[CFG+23] Ran Cohen, Pouyan Forghani, Juan A. Garay, Rutvik Patel, and Vassilis Zikas. Concurrent asynchronous byzantine agreement in expected-constant rounds, revisited. In Guy N. Rothblum and Hoeteck Wee, editors, *Theory of Cryptography - 21st International Conference, TCC 2023*, volume 14372 of *Lecture Notes in Computer Science*, pages 422–451. Springer, 2023.

[CGHZ16] Sandro Coretti, Juan A. Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multi-party computation based on one-way functions. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security*, volume 10032 of *Lecture Notes in Computer Science*, pages 998–1021, 2016.

[CHL21] Annick Chopard, Martin Hirt, and Chen-Da Liu-Zhang. On communication-efficient asynchronous MPC with adaptive security. In Kobbi Nissim and Brent Waters, editors, *Theory of Cryptography - 19th International Conference, TCC 2021*, volume 13043 of *Lecture Notes in Computer Science*, pages 35–65. Springer, 2021.

[CKS05] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptol.*, 18(3):219–246, 2005.

[Coh16] Ran Cohen. Asynchronous secure multiparty computation in constant time. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography*, volume 9615 of *Lecture Notes in Computer Science*, pages 183–207. Springer, 2016.

[CP15] Ashish Choudhury and Arpita Patra. Optimally resilient asynchronous MPC with linear communication complexity. In Sajal K. Das, Dilip Krishnaswamy, Santonu Karkar, Amos Korman, Mohan J. Kumar, Marius Portmann, and Srikanth Sastry, editors, *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015*, pages 5:1–5:10. ACM, 2015.

[CP23] Ashish Choudhury and Arpita Patra. On the communication efficiency of statistically secure asynchronous MPC with optimal resilience. *J. Cryptol.*, 36(2):13, 2023.

[CR93] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 42–51. ACM, 1993.

[CT05] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In Pierre Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005*, volume 3724 of *Lecture Notes in Computer Science*, pages 503–504. Springer, 2005.

[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*, volume 3621 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2005.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590. Springer, 2007.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009*, pages 169–178. ACM, 2009.

[GLM+25] Vipul Goyal, Junru Li, Ankit Kumar Misra, Rafail Ostrovsky, Yifan Song, and Chenkai Weng. Dishonest majority constant-round mpc with linear communication from ddh. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024*, pages 167–199, 2025.

[GLOS25] Vipul Goyal, Junru Li, Rafail Ostrovsky, and Yifan Song. Towards building scalable constant-round MPC from minimal assumptions via round collapsing. *CRYPTO 2025*, 2025.

[GLS24] Vipul Goyal, Chen-Da Liu-Zhang, and Yifan Song. Towards achieving asynchronous MPC with linear communication and optimal resilience. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference*, volume 14927 of *Lecture Notes in Computer Science*, pages 170–206. Springer, 2024.

[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*, pages 218–229. ACM, 1987.

[GPS21] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Unconditional communication-efficient MPC via hall's marriage theorem. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021*, volume 12826 of *Lecture Notes in Computer Science*, pages 275–304. Springer, 2021.

[GPS22] Vipul Goyal, Antigoni Polychroniadou, and Yifan Song. Sharing transformation and dishonest majority MPC with packed secret sharing. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022*, volume 13510 of *Lecture Notes in Computer Science*, pages 3–32. Springer, 2022.

[HNP05] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience (extended abstract). In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 3494 of *Lecture Notes in Computer Science*, pages 322–340. Springer, 2005.

[HNP08] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Asynchronous multi-party computation with quadratic communication. 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*, volume 5126 of *Lecture Notes in Computer Science*, pages 473–485. Springer, 2008.

[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*, 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*, 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*, volume 10624 of *Lecture Notes in Computer Science*, pages 598–628. Springer, 2017.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 145–161, 2003.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing*, pages 21–30. ACM, 2007.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David A. Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591. Springer, 2008.

[JLS24] Xiaoyu Ji, Junru Li, and Yifan Song. Linear-communication asynchronous complete secret sharing with optimal resilience. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference*, volume 14927 of *Lecture Notes in Computer Science*, pages 418–453. Springer, 2024.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013*, volume 7785 of *Lecture Notes in Computer Science*, pages 477–498. Springer, 2013.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference*, volume 9215 of *Lecture Notes in Computer Science*, pages 724–741. Springer, 2015.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. 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*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[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*, volume 9216 of *Lecture Notes in Computer Science*, pages 319–338. Springer, 2015.

[PCR09] Arpita Patra, Ashish Choudhary, and C. Pandu Rangan. Efficient statistical asynchronous verifiable secret sharing with optimal resilience. In Kaoru Kurosawa, editor, *Information Theoretic Security, 4th International Conference, ICITS 2009*, volume 5973 of *Lecture Notes in Computer Science*, pages 74–92. Springer, 2009.

[PS21] Antigoni Polychroniadou and Yifan Song. Constant-overhead unconditionally secure multiparty computation over binary fields. In Anne Canteaut and Franccois-Xavier Standaert, editors,

Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, volume 12697 of *Lecture Notes in Computer Science*, pages 812–841. Springer, 2021.

[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*, pages 73–85. ACM, 1989.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164. IEEE Computer Society, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 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*, pages 1627–1646. ACM, 2020.

A Security Model

In this section, we give more details about the security model.

Real World. In the real world, there exists a set of n parties $P_1, \ldots, P_n$, an adversary $\mathcal{A}$, and an environment $\mathcal{Z}$. The environment provides inputs to the honest parties, receives their outputs, and communicates with the adversary $\mathcal{A}$. We consider $\mathcal{A}$ to be *fully-malicious*. The adversary can corrupt up to t parties and completely control the behavior of the corrupted parties. The parties not controlled by $\mathcal{A}$ are called honest.

The parties and the adversary are modeled as *interactive Turing machines* (ITM), initialized with the random coins and their possible inputs. The protocol proceeds by a sequence of *activations*, where at each point only a single ITM is active. When a party is activated, it can perform local computation and output or send messages to other parties. And if the adversary is activated, it can send messages on behalf of the corrupted parties.

Parties have access to a network of point-to-point asynchronous and secure channels. Asynchronous channels guarantee *eventual* delivery [CR93], meaning that messages sent are eventually delivered. To model the worst-case scenario, the adversary is given the provision to decide the arrival time of each message exchanged between the parties. The adversary cannot drop, change, or inject messages from honest parties. Such channels have been modelized in UC using the *eventual-delivery secure message-transmission* ideal functionality, for example in [KMTZ13, CGHZ16]. The protocol completes once $\mathcal{Z}$ outputs a single bit.

We denote by $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\kappa, z, \bar{r})$ the random variable containing the output of $\mathcal{Z}$ with input z, security parameter κ, and interacting with the parties $P_1, \ldots, P_n$ and the adversary A with random tapes $\bar{r} = (r_1, \ldots, r_n, r_{\mathcal{A}}, r_{\mathcal{Z}})$. We denote the random variable $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\kappa, z, \bar{r})$ for uniformly random $\bar{r}$ by $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\kappa, z)$.

Ideal World. In the ideal world, there exists n dummy parties, a *simulator/ideal adversary* Sim, an environment $\mathcal{Z}$ and the *trusted party/ideal functionality* $\mathcal{F}$. The environment gives inputs to the honest parties, receives outputs, and also interacts with the ideal adversary. As before, the computation finishes once $\mathcal{Z}$ outputs a single bit.

The ideal functionality $\mathcal{F}$ (Figure 12) models the desired behavior of the computation. Sim cannot see or delay the communication between the honest parties and $\mathcal{F}$. In order to model the fact that the adversary can decide when each honest party learns the output, we follow [KMTZ13] and model time via activations. We use a *request-based delay output* to model the output delivery from $\mathcal{F}$ to the honest parties, which is used in [Coh16, CFG+23]. In this model, the functionality $\mathcal{F}$ doesn't directly send the output to the honest parties. Instead, honest parties need to send a "request" to the functionality to get the output. Moreover, the adversary can instruct $\mathcal{F}$ to delay the output for each party by ignoring the corresponding requests. The output can only be delayed for a polynomial number of times, which ensures that the output will eventually be delivered if an honest party sends sufficiently many requests. We consider security with abort, where Sim can decide whether an honest party will receive a correct output from the $\mathcal{F}$ or only receive an Abort signal.

Functionality $\mathcal{F}$

The trusted party interacts with parties $P_1, \ldots, P_n$ and the ideal adversary Sim. For each party P_n, the trusted party initializes an input value $x^{(i)} = \perp$ and output value $y^{(i)} = \perp$.

Public Function: $f : (\{0,1\}^* \cup \{\perp\})^n \to (\{0,1\}^* \cup \{\perp\})^n$.

1. Upon receiving an input v from P_i, if $\mathsf{CoreSet}$ has not been recorded yet or if $P_i \in \mathsf{CoreSet}$, the trusted party sets $x^{(i)} = v$.
2. Upon receiving an input $\mathsf{CoreSet}$ from Sim, the trusted party verifies that $\mathsf{CoreSet}$ is a subset of $\mathcal{P} = \{P_1, \ldots, P_n\}$ of size at least $n - t$, and the inputs of all the honest parties in $\mathsf{Coreset}$ have all been received. If not, the trusted party ignores the message. If $\mathsf{CoreSet}$ has not been recorded yet, then the trusted party records $\mathsf{CoreSet}$ and for every $P_i \notin \mathsf{CoreSet}$, the trusted party sets $x^{(i)} = 0$.
3. If the $\mathsf{CoreSet}$ has been recorded and the value $x^{(i)}$ has been set to a value different from $\perp$ for every

$P_i \in \mathsf{CoreSet}$, then the trusted party computes $(y^{(1)}, \ldots, y^{(m)}) = f(x^{(1)}, \ldots, x^{(m)})$. For each corrupted party P_i, the trusted party sends $y^{(i)}$ to Sim. For each honest party P_i, the trusted party sends $y^{(i)}$ as a request-based delayed output to P_i.

- Upon receiving (abort, P_i) for some honest party P_i, if the output for P_i has not been delivered, the trusted party changes the output of P_i by Abort. Otherwise, the trusted party ignores this request.

Figure 12: Functionality for asynchronous MPC.

We denote by $\mathsf{IDEAL}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}(\kappa, z, \bar{r})$ the random variable containing the output of $\mathcal{Z}$ with input z, security parameter κ, and interacting with the parties $P_1, \ldots, P_n$ and the adversary S with random tapes $\bar{r} = (r_{\mathsf{Sim}}, r_{\mathcal{Z}})$. We denote the random variable $\mathsf{IDEAL}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}(\kappa, z, \bar{r})$ for uniformly random $\bar{r}$ by $\mathsf{IDEAL}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}(\kappa, z)$.

Security. We say Π t-securely realizes $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a probabilistic polynomial time (PPT) simulator Sim in the ideal model such that for any adversary controlling up to t parties and any environment $\mathcal{Z}$, it holds that:

$$\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\kappa, z) \equiv \mathsf{IDEAL}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}(\kappa, z),$$

which means the real-world execution and the ideal-world execution have the same output distribution.

We say Π t-securely realizes $\mathcal{F}$ with statistical security if for any adversary $\mathcal{A}$ there exists a PPT simulator Sim in the ideal model such that for any adversary controlling up to t parties and any environment $\mathcal{Z}$, it holds that:

$$\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\kappa, z) \equiv_{\epsilon} \mathsf{IDEAL}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}(\kappa, z),$$

which means the output distributions of the real-world execution and the ideal-world execution are statistically close, i.e., the total variation distance between the two distributions is no more than $\epsilon = \mathsf{negl}(\kappa)$.

We say Π t-securely realizes $\mathcal{F}$ with computational security if for any adversary $\mathcal{A}$ there exists a PPT simulator Sim in the ideal model such that for any adversary controlling up to t parties and any environment $\mathcal{Z}$, it holds that:

$$\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\kappa, z) \equiv_{c} \mathsf{IDEAL}_{\mathcal{F},\mathsf{Sim},\mathcal{Z}}(\kappa, z),$$

which means the output distributions of the real-world execution and the ideal-world execution are computationally indistinguishable.

The Hybrid Model. In a $\mathcal{G}$-hybrid model, a protocol execution proceeds as in the real world except that the parties have access to an ideal functionality $\mathcal{G}$ for some specific task. During the protocol execution, the parties can communicate with $\mathcal{G}$ as in the ideal world. The UC framework guarantees that an ideal functionality in a hybrid model can be replaced with a protocol that UC-securely realizes $\mathcal{G}$. This is guaranteed by the following composition theorem from [Can01, Can20].

Theorem 6. ([Can01, Can20]) *Let Π be a protocol that UC-securely realizes a functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model and let ρ be a protocol that UC-securely realizes $\mathcal{G}$. Moreover, let Π^{ρ} denote the protocol that is obtained from Π by replacing every ideal call to $\mathcal{G}$ with the protocol ρ. Then protocol Π^{ρ} UC-securely realizes $\mathcal{F}$ in the model where the parties do not have access to the ideal functionality $\mathcal{G}$.*

B Formal Functionalities for Asynchronous Subprotocols

The Functionality $\mathcal{F}_{\mathsf{ABA}}$. The functionality $\mathcal{F}_{\mathsf{ABA}}$ of asynchronous Byzantine agreements is given as follows.

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

The trusted party interacts with parties $P_1, \ldots, P_n$ and the ideal adversary Sim. Let $\mathcal{I} = \mathcal{H}$, where $\mathcal{H}$ is the set of honest parties. For each party P_i, the trusted party initializes x_i and y_i to $\perp$.

1. The trusted party receives a set $\mathcal{P}'$ of parties from Sim, with $|\mathcal{P}'| \leq t$, if no party has received output, then the trusted party sets $\mathcal{I} = \mathcal{H} \backslash \mathcal{P}'$.

2. Upon receiving a message M from P_i:
   (a) If any party or Sim has received an output y, then the trusted party ignores this message; otherwise, the trusted party sets $x_i = M$.
   (b) If $x_i \neq \perp$ for every $P_i \in \mathcal{I}$, then the trusted party sets $y_j = y$ for every $j \in \{1, \ldots, n\}$, where $y = x$ if all inputs $x_j = x$ for $P_j \in \mathcal{I}$, for some $x \neq \perp$. Otherwise, the trusted party sets $y = x_j$ for $P_j \notin \mathcal{H}$ with the smallest index.
   (c) The trusted party sends M to Sim.
3. When the output y_i is set to be some value y, the functionality outputs y as a request-based delayed output to P_i.

Figure 13: Functionality for asynchronous Byzantine agreement.

The Functionality $\mathcal{F}_{\mathsf{AVID}}$. The functionality $\mathcal{F}_{\mathsf{AVID}}$ of Asynchronous Verifiable Information Dispersal is given as follows.

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

The trusted party interacts with all the parties $P_1, \ldots, P_n$, a dealer D and the adversary Sim.

1. Upon receiving a message M from D, the trusted party sends a request-based delayed message $\mathsf{Dispersed}$ to all parties and sends M to Sim.
2. Upon receiving $(\mathsf{Retrieve}, R)$ from $t+1$ parties, the trusted party sends M as the request-based delayed output to R if he has sent $\mathsf{Dispersed}$ before.

Figure 14: Functionality for asynchronous verifiable information dispersal.

The Functionality $\mathcal{F}_{\mathsf{Coin}}$. The functionality $\mathcal{F}_{\mathsf{Coin}}$ of generating a random coin is given as follows.

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

The trusted party interacts with all the parties $P_1, \ldots, P_n$ and the adversary Sim.

1. On receiving $\mathsf{RandCoin}$ from $2t+1$ parties, the trusted party samples $s \in \mathbb{F}_{2^\kappa}$.
2. The trusted party sends s to Sim and sends s to all the honest parties as a request-based delayed output.

Figure 15: Functionality for preparing a random coin.

C Agree on a Common Set

The agreement on a common subset (ACS) primitive allows the parties to agree on a set of at least $n-t$ parties that satisfy a certain property (a so-called ACS property). We give the formal definitions from [BCG93] here.

Definition 1. *Let $m, M \in \mathbb{N}$, and let $\mathcal{U}_1 \ldots \mathcal{U}_n \subseteq [M]$ be a collection of accumulative sets, so that party P_i has $\mathcal{U}_i$. We say that the collection is (m,t)-uniform if the following hold for every malicious corruption of up to t parties:*

- *Every honest party P_i will eventually have $|\mathcal{U}_i| \geq m$.*
- *Every two honest parties P_i and P_j will eventually have $\mathcal{U}_i = \mathcal{U}_j$.*

Definition 2. *Let $m, M \in \mathbb{N}$, and let π be a protocol where the input of each party P_i is an accumulative set $\mathcal{U}_i$. Protocol π is a t-resilient protocol for Agreement on a Common Set (ACS) for n parties (with parameter (M, m) where $m < M$), if the following hold, for every malicious corruption of up to t parties:*

- ***Termination:*** *If the collection $\mathcal{U}_1 \dots \mathcal{U}_n$ is (m,t)-uniform, then with probability 1, all the honest parties eventually complete the protocol.*
- ***Correctness:*** *All the honest parties complete the protocol with a common output $C \subseteq [M]$ so that $|C| \geq m$. Furthermore, every honest party has $C \subseteq \mathcal{U}_i^*$, where $\mathcal{U}_i^*$ is the value of $\mathcal{U}_i$ upon the completion of the protocol.*

We also describe a functionality $\mathcal{F}_{\mathsf{ACS}}$ with parameter (M, m) for ACS as follows.

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

The trusted party interacts with all the parties $P_1, \dots, P_n$ and the adversary Sim with parameter (M, m).

1. The trusted party initializes $S_i = \emptyset \subseteq \{1, \dots, M\}$ for each $i = 1, \dots, n$.
2. Upon receiving an index $k \in \{1, \dots, M\}$ from P_i, the trusted party adds index k to S_i and then forwards k to Sim. If $|\cup_{i=1}^n S_i| \geq m$, the trusted party sets S to be $\cup_{i=1}^n S_i$ and outputs it to all parties as a request-based delayed output.
3. Upon receiving S' from Sim, if a request-based delayed output has been delivered to an honest party, the trusted party ignores the message. Otherwise, the trusted party checks that $|S'| \geq m$, and that for every $k \in S'$, there is some honest party that has input k. If so, the trusted party stops sending the request-based delayed outputs that were sent before, sets $S = S'$, and outputs it to all parties as a request-based delayed output.

Figure 16: Functionality for agree on a common set.

As observed in [CFG+23], by using the "Gather" protocol described in [AJM+23] to do preprocessing and then run M instances of ABA in parallel as in the ACS protocol of [BCG93], $\mathcal{F}_{\mathsf{ACS}}$ can be realized in a constant number of rounds in the $\mathcal{F}_{\mathsf{parABA}}$-hybrid model (see Figure 17 for the functionality). Using the expected constant-time instantiation of $\mathcal{F}_{\mathsf{parABA}}$ in [CFG+23], we get an expected constant-time instantiation of $\mathcal{F}_{\mathsf{ACS}}$.

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

The trusted party is parameterized with a public integer M, and it interacts with parties $P_1, \dots, P_n$ and the ideal adversary Sim. Let $\mathcal{I}_1 = \dots = \mathcal{I}_M = \mathcal{H}$, where $\mathcal{H}$ is the set of honest parties. For each party P_i, the trusted party initializes $x_i^{(1)}, \dots, x_i^{(M)}$ and $y_i^{(1)}, \dots, y_i^{(M)}$ to $\perp$.

1. For each $i = 1, \dots, M$, the trusted party receives a set $\mathcal{P}_i'$ of parties from Sim, with $|\mathcal{P}_i'| \leq t$, if no party has received output, then the trusted party sets $\mathcal{I}_i = \mathcal{H} \backslash \mathcal{P}_i'$.
2. Upon receiving a message (u, j) from P_i:
   (a) If any party or Sim has received output $y_\alpha^{(j)}$ for some $\alpha \in \{1, \dots, n\}$, then the trusted party ignores this message; otherwise, the trusted party sets $y_\alpha^{(j)} = u$ for $\alpha = 1, \dots, n$.
   (b) If $x_\alpha^{(j)} \neq \perp$ for every $P_\alpha \in \mathcal{I}$, then the trusted party sets $y_\alpha^{(j)} = x$ if all inputs $x_\alpha^{(j)} = x$ for $P_\alpha \in \mathcal{I}$, for some $x \neq \perp$. Otherwise, the trusted party sets $y = x_\alpha^{(j)}$ for $P_\alpha \notin \mathcal{H}$ with the smallest index.
   (c) The trusted party sends u to Sim.
3. For each $j = 1, \dots, M$, when the output $y_\alpha^{(j)}$ is set to be some value $y^{(j)}$, the functionality outputs $(y^{(j)}, j)$ as a request-based delayed output to P_i.

Figure 17: Functionality for M parallel ABAs.

We give the theorem of the instantiation for $\mathcal{F}_{\mathsf{ACS}}$ below.

Theorem 7. *There exists an expected constant-time protocol Π_{ACS} that securely realizes $\mathcal{F}_{\mathsf{ACS}}$ with communication of $O(\mathsf{poly}(n, M, \kappa))$ bits.*

D Definition and Instantiation of the LSSSs

We borrow the definition of LSSS from [GLOS25].

Definition 3. (Projection Maps). *Let $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in (\mathbb{F}_q^\ell)^n$. Let $A \subset \{1, \ldots, n\}$ be a non-empty set. The projection map $\pi_A : (\mathbb{F}_q^\ell)^n \to (\mathbb{F}_q^\ell)^{|A|}$ is defined by $\pi_A(\boldsymbol{x}) = (\boldsymbol{x}_i)_{i \in A}$.*

Definition 4. (Linear Secret Sharing Schemes) *Let $\mathbb{F}_q$ be a finite field, and let k, ℓ, and $t < n$ be positive integers. An (n, t, k, ℓ)-linear secret sharing scheme (LSSS) Σ over $\mathbb{F}_q$ consists of two deterministic algorithms $\Sigma.\mathsf{Sh}(\cdot,\cdot) : \mathbb{F}_q^k \times \mathbb{F}_q^{n\ell} \to (\mathbb{F}_q^\ell)^n$ and $\Sigma.\mathsf{Rec}(\cdot) : (\mathbb{F}_q^\ell)^n \to \mathbb{F}_q^k$. For every $\boldsymbol{s} \in \mathbb{F}_q^k$ and $\boldsymbol{r} \in \mathbb{F}_q^{n\ell}$, $\Sigma.\mathsf{Sh}(\boldsymbol{s}, \boldsymbol{r})$ is a linear function that outputs a vector of shares $(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n) \in (\mathbb{F}_q^\ell)^n$. For any $\boldsymbol{c} \in (\mathbb{F}_q^\ell)^n$ which can be outputted by $\Sigma.\mathsf{Sh}(\boldsymbol{s}, \boldsymbol{r})$ for some $\boldsymbol{r} \in \mathbb{F}_q^{n\ell}$, we call $\boldsymbol{c}$ a Σ-sharing of $\boldsymbol{s}$. We require the following three properties.*

- *t-privacy: For all $\boldsymbol{s}, \boldsymbol{s}' \in \mathbb{F}_q^k$ and $A \subset \{1, \ldots, n\}$ of size $\leq t$,*

$$\{\boldsymbol{r} \xleftarrow{\$} \mathbb{F}_q^{n\ell}, \boldsymbol{c} \leftarrow \Sigma.\mathsf{Sh}(\boldsymbol{s}, \boldsymbol{r}) : \pi_A(\boldsymbol{c})\} \equiv \{\boldsymbol{r}' \xleftarrow{\$} \mathbb{F}_q^{n\ell}, \boldsymbol{c}' \leftarrow \Sigma.\mathsf{Sh}(\boldsymbol{s}', \boldsymbol{r}') : \pi_A(\boldsymbol{c}')\}.$$

- *Reconstruction: For every $\boldsymbol{s} \in \mathbb{F}_q^k$, it holds that for any Σ-sharing $\boldsymbol{c}$ of $\boldsymbol{s}$, $\Sigma.\mathsf{Rec}(\boldsymbol{c}) = \boldsymbol{s}$.*
- *Linearity: The two algorithms $\Sigma.\mathsf{Sh}(\cdot,\cdot) : \mathbb{F}_q^k \times \mathbb{F}_q^{n\ell} \to (\mathbb{F}_q^\ell)^n$ and $\Sigma.\mathsf{Rec}(\cdot) : \mathbb{F}_q^{n\ell} \to \mathbb{F}_q^k$ are both $\mathbb{F}_q$-linear.*

we let $\Sigma.\mathsf{Sh}_i(\boldsymbol{s}, \boldsymbol{r}) = \pi_{\{i\}}(\Sigma.\mathsf{Sh}(\boldsymbol{s}, \boldsymbol{r}))$. The LSSS Σ is based on algebraic geometry. For basic knowledge of algebraic geometry, we refer the readers to [CC06]. We briefly introduce the LSSS Σ here. Σ is based on another LSSS Σ'. N parties first agree on a smooth projective absolutely irreducible curve C with genus g defined over $\mathbb{F}_q$ and distinct $\mathbb{F}_q$-rational points

$$Q, P_{-1}, \ldots, P_{-k}, P_1, \ldots, P_N \in C(\mathbb{F}_q).$$

For a divisor D defined by $D = (2g + T) \cdot (Q)$, the sharing algorithm randomly selects $f \in \mathcal{L}(D)$ subject to

$$(f(P_{-1}), \ldots, f(P_{-k})) = \boldsymbol{s}.$$

Then the Σ' secret sharing is defined by

$$[\boldsymbol{s}]' = \Sigma'.\mathsf{Sh}(\boldsymbol{s}, r) = (f(P_1), \ldots, f(P_N)) \in \mathbb{F}_q^N.$$

Now we borrow a lemma from [CC06].

Lemma 1. ([CC06]). *Let E be a divisor on a smooth, projective, absolutely irreducible curve C that is defined over $\mathbb{F}_q$, and suppose that $\ell(E) > 0$. Then each $f \in \mathcal{L}(E)$ is uniquely determined by evaluations of f on any $\deg(E) + 1$ $\mathbb{F}_q$-rational points on C outside the support of E.*

Followed from the above lemma, the secret $\boldsymbol{s}$ of $[\boldsymbol{s}]$ can be reconstructed from any $\deg(D) + 1$ parties' shares, which implies the reconstruction algorithm $\Sigma'.\mathsf{Rec}$. In [CC06], the authors have shown that Σ' is a $(N, T, k, 1)$-LSSS over $\mathbb{F}_q$.

Defining Σ. If $q = 2^\ell$ (ℓ will be specified later), we can restrict the secret on $\mathbb{F}_2^k$ where each entry of the secret is stored in a subspace of $\mathbb{F}_q$ that is isomorphic to $\mathbb{F}_2$. Formally, there exists a bijective $\mathbb{F}_2$-linear map $\mathsf{Conv} : \mathbb{F}_2^\ell \to \mathbb{F}_q$, and both $\mathsf{Conv}, \mathsf{Conv}^{-1}$ can be efficiently computed. In the following, for $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in (\mathbb{F}_2^\ell)^N$, we use $\mathsf{Conv}(\boldsymbol{x})$ to denote $(\mathsf{Conv}(\boldsymbol{x}_1), \ldots, \mathsf{Conv}(\boldsymbol{x}_N))$. And for $\boldsymbol{y} = (y_1, \ldots, y_N) \in \mathbb{F}_q^N$, we use $\mathsf{Conv}^{-1}(\boldsymbol{y})$ to denote $(\mathsf{Conv}^{-1}(y_1), \ldots, \mathsf{Conv}^{-1}(y_N))$. Σ is defined as follows:

- For a vector of secrets $\boldsymbol{s} \in \mathbb{F}_2^k$,

$$\Sigma.\mathsf{Sh}(\boldsymbol{s}, r) = (\mathsf{Conv}^{-1}(\Sigma'.\mathsf{Sh}_1(\boldsymbol{s}, \mathsf{Conv}(r))), \ldots, \mathsf{Conv}^{-1}(\Sigma'.\mathsf{Sh}_N(\boldsymbol{s}, \mathsf{Conv}(r)))).$$

- For a Σ-sharing $(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_N)$,

$$\Sigma.\mathsf{Rec}(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_N) = \Sigma'.\mathsf{Rec}(\mathsf{Conv}(\boldsymbol{c}_1), \ldots, \mathsf{Conv}(\boldsymbol{c}_N)).$$

Defining $\Sigma^{(2)}$. We follow [GLOS25] to define a $\Sigma^{(2)}$-sharing. We first define an LSSS $\Sigma^{(2)'}$ over $\mathbb{F}_q$, denoted by $[.]^{(2)'}$. $\Sigma^{(2)'}$ is the same as Σ except that the divisor D is replaced by $2D$ (i.e. f is selected from $\mathcal{L}(2D)$). For any $x^{(1)}, x^{(2)} \in \mathbb{F}_q$, let $\mathsf{Conv}_2 : \mathbb{F}_2^{\ell^2} \to \mathbb{F}_q$ be a $\mathbb{F}_2$-linear function that maps $\mathsf{Conv}^{-1}(x^{(1)}) \otimes \mathsf{Conv}^{-1}(x^{(2)})$ to $x^{(1)} \cdot x^{(2)}$. Let Conv_2^{-1} be a randomized $\mathbb{F}_2$-linear function such that for all $x \in \mathbb{F}_q$, $\mathsf{Conv}_2^{-1}(x)$ outputs a random vector $\boldsymbol{y} \in \mathbb{F}_2^{\ell^2}$ such that $\mathsf{Conv}_2(\boldsymbol{y}) = x$. $\Sigma^{(2)}$ is defined follows:

- For a vector of secrets $\boldsymbol{s} \in \mathbb{F}_2^k$,

$$\begin{aligned}&\Sigma^{(2)}.\mathsf{Sh}(\boldsymbol{s}, r \| r_1 \| \ldots \| r_N)\\ =&(\mathsf{Conv}_2^{-1}(\Sigma^{(2)'}.\mathsf{Sh}_1(\boldsymbol{s}, \mathsf{Conv}(r)), r_1), \ldots, \mathsf{Conv}_2^{-1}(\Sigma^{(2)'}.\mathsf{Sh}_N(\boldsymbol{s}, \mathsf{Conv}(r)), r_N)).\end{aligned}$$

- For a $\Sigma^{(2)}$-sharing $(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_N)$,

$$\Sigma^{(2)}.\mathsf{Rec}(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_N) = \Sigma^{(2)'}.\mathsf{Rec}(\mathsf{Conv}_2(\boldsymbol{c}_1), \ldots, \mathsf{Conv}_2(\boldsymbol{c}_N)).$$

It can be easily verified that the tensor product of two Σ-sharings $[\boldsymbol{s}^{(1)}] = (\boldsymbol{c}_1^{(1)}, \ldots, \boldsymbol{c}_N^{(1)}), [\boldsymbol{s}^{(2)}] = (\boldsymbol{c}_1^{(2)}, \ldots, \boldsymbol{c}_N^{(2)})$ is a $\Sigma^{(2)}$-sharing $[\boldsymbol{s}^{(1)} * \boldsymbol{s}^{(2)}]^{(2)} = (\boldsymbol{c}_1^{(1)} \otimes \boldsymbol{c}_1^{(2)}, \ldots, \boldsymbol{c}_N^{(1)} \otimes \boldsymbol{c}_N^{(2)})$. We can simply write $[\boldsymbol{s}^{(1)}] \otimes [\boldsymbol{s}^{(2)}] = [\boldsymbol{s}^{(1)} * \boldsymbol{s}^{(2)}]^{(2)}$.

Defining $\Sigma^{(3)}$. Similarly, we follow [GLOS25] to define $\Sigma^{(3)}$-sharings. First, $\Sigma^{(3)'}$ is defined over $\mathbb{F}_q$ by replacing D by $3D$ in Σ'. For any $x^{(1)}, x^{(2)}, x^{(3)} \in \mathbb{F}_q$, let $\mathsf{Conv}_3 : \mathbb{F}_2^{\ell^3} \to \mathbb{F}_q$ be a $\mathbb{F}_2$-linear function that maps $\mathsf{Conv}^{-1}(x^{(1)}) \otimes \mathsf{Conv}^{-1}(x^{(2)}) \otimes \mathsf{Conv}^{-1}(x^{(3)})$ to $x^{(1)} \cdot x^{(2)} \cdot x^{(3)}$. Let Conv_3^{-1} be a randomized $\mathbb{F}_2$-linear function such that for all $x \in \mathbb{F}_q$, $\mathsf{Conv}_3^{-1}(x)$ outputs a random vector $\boldsymbol{y} \in \mathbb{F}_2^{\ell^3}$ such that $\mathsf{Conv}_3(\boldsymbol{y}) = x$. $\Sigma^{(3)}$ is defined as follows:

- For a vector of secrets $\boldsymbol{s} \in \mathbb{F}_2^k$,

$$\begin{aligned}&\Sigma^{(3)}.\mathsf{Sh}(\boldsymbol{s}, r \| r_1 \| \ldots \| r_N)\\ =&(\mathsf{Conv}_3^{-1}(\Sigma^{(3)'}.\mathsf{Sh}_1(\boldsymbol{s}, \mathsf{Conv}(r)), r_1), \ldots, \mathsf{Conv}_3^{-1}(\Sigma^{(3)'}.\mathsf{Sh}_N(\boldsymbol{s}, \mathsf{Conv}(r)), r_N)).\end{aligned}$$

- For a $\Sigma^{(3)}$-sharing $(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_N)$,

$$\Sigma^{(3)}.\mathsf{Rec}(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_N) = \Sigma^{(3)'}.\mathsf{Rec}(\mathsf{Conv}_3(\boldsymbol{c}_1), \ldots, \mathsf{Conv}_3(\boldsymbol{c}_N)).$$

The tensor product of a $\Sigma^{(2)}$-sharing and a Σ-sharing is a $\Sigma^{(3)}$-sharing, namely $[\boldsymbol{s}^{(1)}]^{(2)} \otimes [\boldsymbol{s}^{(2)}] = [\boldsymbol{s}^{(1)} * \boldsymbol{s}^{(2)}]^{(3)}$.

Following the result of [CC06], when we choose $T = N/20$, $q = 2^{20}$, $k = N/20$, and $\ell = 20$, the resulting $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$ are all $(N, T, k, 20)$-LSSSs over $\mathbb{F}_2$. In addition, the secret of a $\Sigma^{(3)}$-sharing can be reconstructed from any $N/5$ shares, i.e. for any set $A \subset \{1, \ldots, N\}$ such that $|A| \geq N/5$, there exists an algorithm $\Sigma^{(3)}.\mathsf{Rec}_A : (\mathbb{F}_2^\ell)^{|A|} \to \mathbb{F}_2^k$ such that for any $\boldsymbol{s} \in \mathbb{F}_2^k$ and $\boldsymbol{r} \in \mathbb{F}_2^{N\ell}$:

$$\Sigma^{(3)}.\mathsf{Rec}_A(\pi_A(\Sigma^{(3)}.\mathsf{Sh}(\boldsymbol{s}, \boldsymbol{r}))) = \boldsymbol{s}.$$

This also implies that the secret of a Σ or $\Sigma^{(2)}$-sharing can be reconstructed from $N/5$ shares (we can simply multiply a Σ-sharing $[\mathbf{1}]$ on it to convert them to a $\Sigma^{(3)}$-sharing with the same secret). In addition, for any $N/4$ shares of a $\Sigma^{(3)}$-sharing, if T of the shares were wrong, the other $N/5$ shares uniquely determine the secret.

Remark 3. *As noted in [GLOS25], for a (n, t, k, ℓ)-LSSS Σ, we have the following algorithms. There are algorithms that can efficiently determine whether a set S of shares comes from a valid Σ-sharing and give such a sharing (if it is valid). There are also algorithms that can efficiently sample a set of at most t_1 shares from another t_2 share ($t_1 + t_2 \leq T$) of a Σ-sharing without the secret. We refer the readers to [GLOS25] for more details.*

E Subprotocols for Π_0

Preparing Random $\Sigma^{(2)}$-Sharings. We need to prepare random $\Sigma^{(2)}$-sharings in the preprocessing phase, we follow the approach from [PS21]. Take $a = \lfloor \log N \rfloor + 1$, and let $\Sigma_{\times a}, \Sigma^{(2)}_{\times a}$ be the a-fold interleaved secret sharing of $\Sigma, \Sigma^{(2)}$ respectively. Let $[\cdot]_a, [\cdot]^{(2)}_a$ denote sharings in $\Sigma_{\times a}, \Sigma^{(2)}_{\times a}$. The servers run the following protocol $\Pi_{\mathsf{RandShare}}$.

Protocol $\Pi_{\mathsf{RandShare}}$

1. Each server S_i samples a random $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{s}_i]^{(2)}_a$ and distributes it to all the servers.
2. Let $t = n/4$, $\mathcal{V}$ be the matrix
$$\mathcal{V} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & b_1 & \cdots & b_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & b_1^{n-t-1} & \cdots & b_{n-1}^{n-t-1} \end{pmatrix},$$
where $1, b_1, \ldots, b_{n-1}$ are n different elements in $\mathbb{F}_{2^a}$. The servers locally compute
$$\begin{pmatrix} [\boldsymbol{r}_1]^{(2)}_a \\ [\boldsymbol{r}_2]^{(2)}_a \\ \vdots \\ [\boldsymbol{r}_{n-t}]^{(2)}_a \end{pmatrix} = \mathcal{V} \cdot \begin{pmatrix} [\boldsymbol{s}_1]^{(2)}_a \\ [\boldsymbol{s}_2]^{(2)}_a \\ \vdots \\ [\boldsymbol{s}_n]^{(2)}_a \end{pmatrix}.$$
3. Each $\Sigma^{(2)}_{\times a}$-sharing
$$[\boldsymbol{r}_i]^{(2)}_a = ([\boldsymbol{r}_i^{(1)}]^{(2)}, \ldots, [\boldsymbol{r}_i^{(a)}]^{(2)}).$$
Thus the parties obtain $a \cdot (n-t)$ random $\Sigma^{(2)}$-sharings $[\boldsymbol{r}_i^{(j)}]^{(2)}$ for $i = 1, \ldots, n-t, j = 1, \ldots, a$.

Figure 18: Preparing random $\Sigma^{(2)}$-sharings.

In this way, each random $\Sigma^{(2)}$-sharing can be prepared with amortized cost $O(N)$.

The Transpose Protocol. We first present the protocol $\Pi_{\mathsf{Transpose}}$ [GLOS25] which is used for degree reduction. $\Pi_{\mathsf{Transpose}}$ takes sharings $[\boldsymbol{x}_1]^{(2)}, \ldots, [\boldsymbol{x}_k]^{(2)}$ as input. Let $\boldsymbol{x}_i = (x_{i,1}, \ldots, x_{i,k})$ and $\boldsymbol{x}_i^* = (x_{1,i}, \ldots, x_{k,i})$, $\Pi_{\mathsf{Transpose}}$ outputs $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$. The degree reduction process uses $\Pi_{\mathsf{Transpose}}$ as a building block.

For each execution of $\Pi_{\mathsf{Transpose}}$, the servers prepare mask sharings for the $\Sigma^{(2)}$-sharings to be reconstructed, and also generate randomness by generating $N\ell$ $\Sigma^{(2)}$-sharings for column vectors. We let the $N\ell$ random secrets in $\mathbb{F}_2^k$ of the $\Sigma^{(2)}$ serve as the randomness (in $\mathbb{F}_2^{N\ell}$) of $\Sigma.\mathsf{Sh}$ for generating the k column Σ-sharings.

Since the sharing algorithm Sh of a LSSS is linear on each field element of its input, $\Sigma.\mathsf{Sh}_1, \ldots, \Sigma.\mathsf{Sh}_N : \mathbb{F}_2^{k+N\ell} \to \mathbb{F}_2^{\ell}$ are all $\mathbb{F}_2$-linear functions. Suppose that for each $i = 1, \ldots, N$,
$$\begin{aligned} &\Sigma.\mathsf{Sh}_i((s_1, \ldots, s_k), (a_1, \ldots, a_{N\ell})) \\ =&\left(\sum_{j=1}^{k} c^{(i)}_{1,j} s_j + \sum_{j=1}^{N\ell} c^{(i)}_{1,k+j} a_j, \ldots, \sum_{j=1}^{k} c^{(i)}_{\ell,j} s_j + \sum_{j=1}^{N\ell} c^{(i)}_{\ell,k+j} a_j \right). \end{aligned}$$

Correspondingly, we define $\mathbb{F}_2$-linear functions $F_1, \ldots, F_N : (\mathbb{F}_2^k)^{k+N\ell} \to (\mathbb{F}_2^k)^{\ell}$ by
$$F_i(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{k+N\ell}) = \left(\sum_{j=1}^{k+N\ell} c^{(i)}_{1,j} \boldsymbol{v}_j, \ldots, \sum_{j=1}^{k+N\ell} c^{(i)}_{\ell,j} \boldsymbol{v}_j \right)$$

for each $i = 1, \ldots, N$ and give $\Pi_{\mathsf{Transpose}}$ as follows.

Protocol $\Pi_{\mathsf{Transpose}}$

Input: Each server's shares of input sharings $[\boldsymbol{x}_1]^{(2)}, \ldots, [\boldsymbol{x}_k]^{(2)}$, a group of N random $\Sigma^{(2)}$-sharings $[\boldsymbol{r}_1]^{(2)}, \ldots, [\boldsymbol{r}_N]^{(2)}$ where each $\boldsymbol{r}_j$ is generated by S_j, and $N\ell$ $\Sigma^{(2)}$-sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{N\ell}]^{(2)}$. Let each $\boldsymbol{x}_i = (x_{i,1}, \ldots, x_{i,k})$ and set $\boldsymbol{x}_i^* = (x_{1,i}, \ldots, x_{k,i})$.

1. The servers locally compute $[\boldsymbol{y}_i]^{(2)} = [F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{N\ell}) + \boldsymbol{r}_i]^{(2)}$ for $i = 1, \ldots, N$.
2. For each $i = 1, \ldots, N$, each server sends his share of $[\boldsymbol{y}_i]^{(2)}$ to server S_i for reconstruction.
3. Each server S_j reconstructs $\boldsymbol{y}_j$ and computes the vector of his shares of $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$ by $\boldsymbol{y}_i - \boldsymbol{r}_i$.

Figure 19: Protocol to turn row $\Sigma^{(2)}$-sharings to column Σ-sharings.

Now, we show how to utilize $\Pi_{\mathsf{Transpose}}$ to compute groups of multiplication gates, do sharing transformations, and collect secrets from different positions.

Batched Multiplication Gates. For each k batches of k multiplication gates (altogether k^2 multiplication gates evaluated together), we use the following protocol Π_{Multi} to evaluate them. The communication cost of Π_{Multi} is $O(N^2)$.

Protocol Π_{Multi}

Input: The servers input their shares of Σ-sharings $[\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_k]$ and $[\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_k]$. Let $\boldsymbol{z}_1 = \boldsymbol{x}_1 * \boldsymbol{y}_1, \ldots, \boldsymbol{z}_k = \boldsymbol{x}_k * \boldsymbol{y}_k$.

1. The servers locally computes their shares of $[\boldsymbol{z}_j]^{(2)} = [\boldsymbol{x}_j] \otimes [\boldsymbol{y}_j]$ for each $j = 1, \ldots, k$.
2. The servers run $\Pi_{\mathsf{Transpose}}$ with input sharings $[\boldsymbol{z}_1]^{(2)}, \ldots, [\boldsymbol{z}_k]^{(2)}$ and get output sharings $[\boldsymbol{z}_1^*], \ldots, [\boldsymbol{z}_k^*]$.
3. The servers locally computes $[\boldsymbol{z}_j^*]^{(2)} = [\mathbf{1}] \otimes [\boldsymbol{z}_j^*]$ for each $j = 1, \ldots, k$, where $[\mathbf{1}]$ is a public Σ-sharing of an all-1 vector.
4. The servers run $\Pi_{\mathsf{Transpose}}$ with input sharings $[\boldsymbol{z}_1^*]^{(2)}, \ldots, [\boldsymbol{z}_k^*]^{(2)}$ and get output sharings $[\boldsymbol{z}_1], \ldots, [\boldsymbol{z}_k]$.

Figure 20: Protocol to compute batched multiplication gates.

F Chernoff Bound

Let $\mathbb{E}(X)$ denote the expectation of a random variable X, below is a well-known lemma in the probability theory.

Lemma 2. (Chernoff Bound). *Suppose $X_1, \ldots, X_m$ are independent random variables taking values in $\{0, 1\}$, and let $X = X_1 + \cdots + X_m$ be their sum, and $\mathbb{E}(X) = \mu$. Then for any $\delta > 0$ it holds that:*

$$P(X \geq (1+\delta)\mu) \leq e^{-\frac{\delta^2 \mu}{2+\delta}}.$$

Also for any $0 < \delta < 1$ it holds that:

$$P(X \leq (1-\delta)\mu) \leq e^{-\frac{\delta^2 \mu}{2}}.$$

G Subprotocols for the Inner Protocol

When an additive sharing $\langle x \rangle$ is distributed among 3 parties, they can run the following protocol Π_{Open} to open the secret.

Protocol $\Pi_{\mathsf{Open}}(\langle x\rangle)\}$

The protocol runs between 3 parties P_1, P_2, P_3. To open the secret x of $\langle x\rangle$:

1. Each party P_i sends his share of $\langle x\rangle$ to each other party.
2. Each party reconstructs x with all the parties' shares of $\langle x\rangle$.

Figure 21: Protocol for opening the secret of an additive sharing.

To compute a multiplication for a virtual server, the 3 servers that emulate the virtual server run the following protocol Π_{Mult}.

Protocol $\Pi_{\mathsf{Mult}}(\langle \boldsymbol{x}\rangle, \langle y\rangle, \{(r_0^{(i,j)}, r_1^{(i,j)})_{P_i}, (b^{(i,j)}, r_{b^{(i,j)}}^{(i,j)})_{P_j}\}_{i\neq j\in\{1,2,3\}})$

The protocol runs between 3 parties P_1, P_2, P_3. To compute $\langle \boldsymbol{z}\rangle = \langle \boldsymbol{x}\cdot y\rangle$ ($\boldsymbol{x}\in\mathbb{F}_2^{\kappa-1}, y\in\mathbb{F}_2$) with each pair of parties (P_i, P_j) holding the result from an invocation of $\mathcal{F}_{\mathsf{ROT}}$, i.e. P_i holds a pair of random strings $r_0^{(i,j)}, r_1^{(i,j)}$ and P_j holds $b^{(i,j)}, r_{b^{(i,j)}}^{(i,j)}$:

1. Let each party P_i's shares of $\langle \boldsymbol{x}\rangle, \langle y\rangle$ be $x^{(i)}, y^{(i)}$ respectively. For each pair of parties (P_i, P_j):
   (a) P_i sends $r_1^{(i,j)} \oplus r_0^{(i,j)} \oplus x^{(i)}$ to P_j. P_j sends $y^{(j)} \oplus b^{(i,j)}$ to P_i.
   (b) P_i locally computes $z_i^{(i,j)} = r_{y^{(j)}\oplus b^{(i,j)}}^{(i,j)}$. P_j locally computes $z_j^{(i,j)} = (r_1^{(i,j)} \oplus r_0^{(i,j)} \oplus x^{(i)})\cdot y^{(j)} \oplus r_{b^{(i,j)}}^{(i,j)}$. Then we have
   $$z_i^{(i,j)} \oplus z_j^{(i,j)} = x^{(i)}\cdot y^{(j)}.$$
2. Each party P_i computes
   $$z^{(i)} = x^{(i)}\cdot y^{(i)} \oplus \sum_{j\neq i}(z_i^{(i,j)} \oplus z_i^{(j,i)})$$
   as his share of $\langle \boldsymbol{z}\rangle$.

Figure 22: Protocol for multiplication.

H The process of Evaluating the Circuit from the Garbled Circuits

The evaluation process Π_{Ev} of each party P_i is given as follows.

Protocol Π_{Ev}

1. For each $V_j \in \mathsf{Ter}$, P_i separates the circuit $\mathtt{Circ}^{V_j}$ to $\mathtt{Circ}_1^{V_j}, \ldots, \mathtt{Circ}_{\mathsf{rec}}^{V_j}$, each $\mathtt{Circ}_i^{V_j}$ outputs V_j's share of the i-th reconstruction of $\Sigma^{(2)}$-sharings in the evaluation phase of Π_0, i.e. V_j's share of $[\boldsymbol{s}_i]^{(2)}$.
2. For each $V_j \in \mathsf{Ter}$, since no input to $\mathtt{Circ}_1^{V_j}$ comes from reconstruction, P_i already gets the input labels $k_{w,x_w\oplus\lambda_w}^{P_{j,1}}, k_{w,x_w\oplus\lambda_w}^{P_{j,2}}, k_{w,x_w\oplus\lambda_w}^{P_{j,3}}$ and $x_w\oplus\lambda_w$. Then for each gate g (excluding input gates) in $\mathtt{Circ}_1^{V_j}$:
   – If g is an XOR gate with input wires a, b and output wire o, P_i computes $k_{o,x_o\oplus\lambda_o}^{P_{j,\beta}} = k_{a,x_a\oplus\lambda_a}^{P_{j,\beta}} \oplus k_{b,x_b\oplus\lambda_b}^{P_{j,\beta}}$ for each $\beta = 1,2,3$ and $x_o\oplus\lambda_o = (x_a\oplus\lambda_a)\oplus(x_b\oplus\lambda_b)$.
   – If g is an AND gate with input wires a, b and output wire o, P_i computes
   $$\begin{aligned} & k_{o,x_o\oplus\lambda_o}^{P_{j,\beta}} \| (x_o\oplus\lambda_o) \\ =& A_{g,2(x_a\oplus\lambda_a)+x_b\oplus\lambda_b}^{P_{j,\beta}} \\ \oplus& \Big(\bigoplus_{i=1}^{3} \big(\mathcal{O}(k_{a,x_a\oplus\lambda_a}^{P_{j,i}} \| (x_a\oplus\lambda_a) \| k_{b,x_b\oplus\lambda_b}^{P_{j,i}} \| (x_b\oplus\lambda_b) \| i \| j \| \beta \| g)\big) \Big) \end{aligned}$$
   for each $\beta = 1,2,3$.

– If g is an output gate (indexed k) with output wire w, P_i computes

$$Y^{P_{j,\beta}}_{\mathsf{k},x_w} = \mathsf{ct}^{P_{j,\beta}}_{w,x_w\oplus\lambda_w} \oplus \left(\bigoplus_{i=1}^{3} \mathcal{O}(k^{P_{j,i}}_{w,x_w\oplus\lambda_w} \| (x_w \oplus \lambda_w) \| i \| j \| \beta \| w)\right)$$

for each $\beta = 1, 2, 3$.

3. After evaluating all the gate of $\mathtt{Circ}_1^{V_j}$ for every $V_j \in \mathsf{Ter}$, P_i obtains the sharings $[\boldsymbol{s}_1]^{(2)}$ and $\{Y^{P_{j,\beta}}_{a,s^{V_j}_{1,a}}\}_{a=1}^{\ell^2}$ for each $\beta = 1, 2, 3$, where $s^{V_j}_{1,a}$ is the a-th bit of V_j's share of $[\boldsymbol{s}_1]^{(2)}$. Then P_i checks whether the sharings $[\boldsymbol{s}_1]^{(2)}$ and $\{[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}]^{(3)}\}_{\alpha=1}^{\kappa}$ for each $\beta = 1, 2, 3$ are all valid. If not, P_i aborts the protocol. Otherwise, P_i reconstructs $\boldsymbol{s}_1$ and $\{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}\}_{\alpha=1}^{\kappa}$ for each $\beta = 1, 2, 3$.
4. For each $i = 1, \ldots, \mathsf{rec}$, if the receiver of $[\boldsymbol{s}_i]^{(2)}$ is a virtual server $V_j \in \mathsf{Ter}$ and the η-th bit $s_{1,\eta}$ of $\boldsymbol{s}_1$ is used as an input wire with index j_η in circuit $\mathtt{Circ}^{V_j}$, with $\boldsymbol{s}_1$ and $\{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}\}_{\alpha=1}^{\kappa}$, P_i decrypts $k^{P_{j,\beta}}_{w_{j_\eta},s_{1,\eta}\oplus\lambda_{w_{j_\eta}}} \|(s_{1,\eta} \oplus \lambda_{w_{j_\eta}})$ from the ciphertexts $\mathsf{ct}^{(i,\beta)}_{j_\eta,s_{1,\eta}}$ for each $\beta = 1, 2, 3$ by

$$k^{P_{j,\beta}}_{w_{j_\eta},s_{1,\eta}\oplus\lambda_{w_{j_\eta}}} \|(s_{1,\eta} \oplus \lambda_{w_{j_\eta}}) = \mathsf{ct}^{(1,\beta)}_{j_\eta,s_{1,\eta}} \oplus \mathcal{O}\big(\boldsymbol{r}_{s_{1,\eta},\eta,\beta} \| s_{1,\eta} \| 1 \| \beta \| \eta \| j_\eta\big).$$

5. For each $V_j \in \mathsf{Ter}$, now P_i has the input labels $k^{P_{j,1}}_{w,x_w\oplus\lambda_w}, k^{P_{j,2}}_{w,x_w\oplus\lambda_w}, k^{P_{j,3}}_{w,x_w\oplus\lambda_w}$ and $x_w \oplus \lambda_w$ for each input wire w of $\mathtt{Circ}_2^{V_j}$. Thus, P_i can evaluate $\mathtt{Circ}_2^{V_1}, \ldots, \mathtt{Circ}_2^{V_N}$ in the same way as Steps 2-4. Repeating the above steps, P_i eventually obtains all the $\Sigma^{(2)}$-sharings $[\boldsymbol{s}_i]^{(2)}$ whose receiver R_i is a client together with $\{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}\}_{\alpha=1}^{\kappa}$ for each $\beta = 1, 2, 3$ if the protocol is not aborted. Then, P_i reconstructs the secrets of these sharings.

Figure 23: The code of evaluating the circuit for each party P_i.

I Security Proof of the Main Protocol

Proof. We prove the security of Π by constructing an ideal adversary Sim. Sim needs to interact with the environment $\mathcal{Z}$ and with the ideal functionalities. Sim constructs virtual real-world honest parties and runs the real-world adversary $\mathcal{A}$. For simplicity, we just let Sim communicate with $\mathcal{A}$ on behalf of honest parties and the ideal functionality of sub-protocols in our proof. In order to simulate the communication with $\mathcal{Z}$, every message that Sim receives from $\mathcal{Z}$ is sent to $\mathcal{A}$, and likewise, every message sent from $\mathcal{A}$ to $\mathcal{Z}$ is forwarded by Sim. Each time an honest party needs to send a message to another honest party, Sim will tell $\mathcal{A}$ that a message has been delivered such that $\mathcal{A}$ can tell Sim the arrival time of this message to help Sim instruct the functionalities to delay the outputs in the ideal world. For each request-based delayed output that needs to be sent to an honest party, we let Sim delay the output in default until we say Sim allows the functionality to send the output. We will show that the output in the ideal world is identically distributed to that in the real world using hybrid arguments.

Let the set of corrupted parties be $\mathcal{C}$ and the set of honest parties be $\mathcal{H}$. The ideal adversary Sim runs $\mathsf{Sim}_{\mathsf{Setup}}, \mathsf{Sim}_{\mathsf{Share}}, \mathsf{Sim}_{\mathsf{Local}}, \mathsf{Sim}_{\mathsf{Garble}}, \mathsf{Sim}_{\mathsf{Ver}}, \mathsf{Sim}_{\mathsf{Eval}}$ provided below.

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

Setup Phase

Sim faithfully emulates $\mathcal{F}_{\mathsf{Setup}}$ to interact with the adversary. More concretely:

1. **Preparing Symmetric Keys.** For each corrupted party P_i and honest party P_j, Sim samples a random κ-bit string as $k_{i,j} = k_{j,i}$ and sends it to $\mathcal{A}$.
2. **Determining the Virtual Servers.** Sim randomly samples a 3-party set $\{P_{j,1}, P_{j,2}, P_{j,3}\}$ for each $j = 1, \ldots, N$ and sends them to $\mathcal{A}$. If for more than $N/26$ of $j \in \{1, \ldots, N\}$, $P_{j,1}, P_{j,2}, P_{j,3}$ are all corrupted, Sim aborts the simulation.

3. **Preparing the Seeds for Commitments.** Sim receives $q_i^{P_{j,\alpha}}(\alpha_u)$ for each corrupted party P_u, $i = 1, \ldots, n$, $j = 1, \ldots, N$, and $\alpha = 1, 2, 3$ from $\mathcal{A}$. Then, for each corrupted party P_i, Sim sends $\mathsf{seed}_i^{P_{j,\alpha}}$ to $\mathcal{A}$. Sim also receives $q^{P_{i,j}}(\alpha_u), q'^{P_{i,j}}(\alpha_u), q_0^{P_{i,j}}(\alpha_u)$ for each corrupted party P_u, $i = 1, \ldots, N$, and $j = 1, 2, 3$ from $\mathcal{A}$. Then, for each corrupted party $P_{i,j}$, Sim sends $\mathsf{seed}^{P_{i,j}}, \mathsf{seed}'^{P_{i,j}}, \mathsf{seed}_0^{P_{i,j}}$ to $\mathcal{A}$.
4. **Preparing ROT Instances.** For each honest virtual server V_j, for each each pair of parties $(P_{j,\alpha}, P_{j,\beta})$ where $P_{j,\alpha}$ is honest but $P_{j,\beta}$ is corrutped, Sim randomly samples a bit as $b^{(\alpha,\beta)}$ and a random $(\kappa-1)$-bit string as $r_{b^{(1,\beta)}}^{(\alpha,\beta)}$ and emulates $\mathcal{F}_{\mathsf{Setup}}$ to send $(b^{(\alpha,\beta)}, r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)})$ to $\mathcal{A}$. Similarly, for each pair of parties $(P_{j,\alpha}, P_{j,\beta})$ where $P_{j,\alpha}$ is corrupted but $P_{j,\beta}$ is honest, Sim randomly samples $(\kappa-1)$-bit strings $r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)}$ and emulates $\mathcal{F}_{\mathsf{Setup}}$ to send $(r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)})$ to $\mathcal{A}$. for each pair of corrupted servers $(P_{j,\alpha}, P_{j,\beta})$, Sim randomly samples $(\kappa-1)$-bit strings $r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)}$ and the bit $b^{(\alpha,\beta)}$, then Sim emulates $\mathcal{F}_{\mathsf{Setup}}$ to send $(r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)}), (b^{(\alpha,\beta)}, r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)})$ to $\mathcal{A}$.

Additionally, Sim initializes $\mathsf{CompCheck} = \mathsf{CorrCheck} = \mathsf{OTCheck} = 0$. We say a virtual server V_j is honest if there exists an honest party among $P_{j,1}, P_{j,2}, P_{j,3}$. Otherwise, we say the virtual server is corrupted. For each honest virtual server V_j, Sim sets $\mathsf{Corr}_j = \mathsf{Comp}_j = \mathsf{Check}_j = \mathsf{ROT}_j = 0$.

Figure 24: The simulator for the setup phase of Π.

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

Sharing Phase

1. **Preparing Random $\Sigma^{(2)}$-Sharings.** For each execution of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ in this step, for each party P_i, Sim does the following:
   (a) If P_i is honest, Sim samples each $\boldsymbol{v}_{i,\alpha}^{V_j}$ for corrupted party $P_{j,\alpha} = P_{j_\alpha}$ randomly and follows the protocol to compute $\boldsymbol{v}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ and emulates $\mathcal{F}_{\mathsf{AVID}}$ to send it to $\mathcal{A}$. For each honest party $P_{j,\alpha} = P_{j_\alpha}$, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ to send a random message in $\mathbb{F}_2^\ell$ as $\boldsymbol{v}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ to $\mathcal{A}$ for each corrupted party $P_{j,\alpha} = P_{j_\alpha}$ (Sim still emulates $\mathcal{O}$ faithfully, same below unless stated otherwise). If P_i is corrupted, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ to receive $\boldsymbol{v}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ for each $j = 1, \ldots, N$ and $\alpha = 1, 2, 3$. Similarly for $\boldsymbol{o}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$.
   (b) For each instance of $\mathcal{F}_{\mathsf{AVID}}$ emulated in the last step that is used to send a share from P_i to $P_{j,\alpha}$, when the $(\mathsf{Retrieve}, P_{j,\alpha})$ message sent from a honest parties are allowed to be delivered to $\mathcal{F}_{\mathsf{AVID}}$ and Sim receives $(\mathsf{Retrieve}, P_{j,\alpha})$ from b corrupted parties where $a + b \geq t + 1$, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ to send $\boldsymbol{v}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ to $P_{j,\alpha}$ if $P_{j,\alpha}$ is corrupted. Similarly for $\boldsymbol{o}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$.

   For each sharing $[\boldsymbol{v}'_\alpha]^{(2)}$ generated by P_α in an execution of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ that we need to let a party P_i hold the secret $\boldsymbol{v}_\alpha = \boldsymbol{v}'_\alpha$, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ to send a random message in $\mathbb{F}_2^\ell$ as $\boldsymbol{v}_{\alpha,\beta}^{V_j} \oplus \mathcal{O}(k_{\alpha,i,j_\alpha} \| \mathsf{mid})$ to $\mathcal{A}$ for each $\alpha = 1, 2, 3$ and $\beta = 1, \ldots, N$ if P_α is honest. If P_α is corrupted, Sim receives $\boldsymbol{v}_{\alpha,\beta}^{V_j} \oplus \mathcal{O}(k_{\alpha,i,j_\alpha} \| \mathsf{mid})$ for each $\alpha = 1, 2, 3$ and $\beta = 1, \ldots, N$ from $\mathcal{A}$. For each honest party $P_{j,\beta} = P_{j_\beta}$, Sim follows the protocol to check whether his shares obtained from the two messages $\boldsymbol{v}_{\alpha,\beta}^{V_j} \oplus \mathcal{O}(k_{\alpha,i,j_\beta} \| \mathsf{mid})$ and $\boldsymbol{v}_{\alpha,\beta}^{V_j} \oplus \mathcal{O}(k_{\alpha,j_\beta} \| \mathsf{mid})$ are the same. If not, Sim aborts the protocol on behalf of the honest party $P_{j,\beta}$ after the two messages from $\mathcal{F}_{\mathsf{AVID}}$ are delivered to $P_{j,\beta}$. After the simulation terminates, Sim outputs what $\mathcal{A}$ outputs.
2. **Preprocessing for the Verification of Sharings.** For the distribute process of the sharings $[\boldsymbol{r}_1^{(i)}]_{\kappa'}, [\boldsymbol{r}_2^{(i)}]^{(2)}_{\kappa'}, [\boldsymbol{r}_3^{(i)}]^{(3)}_{\kappa'}$ and $[\boldsymbol{o}_1^{(i)}]_{\kappa'}, [\boldsymbol{o}_2^{(i)}]^{(2)}_{\kappa'}, [\boldsymbol{o}_3^{(i)}]^{(3)}_{\kappa'}$ for each party P_i, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ as in Step 1 to interact with $\mathcal{A}$.
3. **Sharing Inputs.** For each batch of input wires attached to P_i, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ as in Step 1 to simulate the distributing process of the input sharing.
4. **Preparing for the Garbling of Local Circuits.** For each execution of $\Pi^{(3)}_{\mathsf{RandShare\text{-}Share}}$ in this step, for each party P_i, Sim emulate $\mathcal{F}_{\mathsf{AVID}}$ to interact with $\mathcal{A}$ as in Step 1 to simulate the process of distributing shares.
5. **Committing Sharings.** For each honest party P_i, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ to send a random message (of the

same length) as $\mathsf{Sh}_i^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}_i^{P_{j,\alpha}} \| \mathsf{mid})$ to $\mathcal{A}$ for each $j = 1, \ldots, N$ and $\alpha = 1, 2, 3$. For each corrupted party P_i, Sim receives $\mathsf{Sh}_i^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}_i^{P_{j,\alpha}} \| \mathsf{mid})$ for each $j = 1, \ldots, N$ and $\alpha = 1, 2, 3$ from $\mathcal{A}$. Sim then computes $\mathsf{Sh}_i^{P_{j,\alpha}}$ with $\mathsf{Sh}_i^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}_i^{P_{j,\alpha}} \| \mathsf{mid})$ and $\mathsf{seed}_i^{P_{j,\alpha}}$.

6. **Determining the Core Set.** For each honest party P_i, Sim considers that P_i likes another party P_j if all the retrieving process of P_i's shares generating by P_j terminates and the $\mathsf{Dispersed}$ message of P_j's commitment of P_i's shares has been delivered to P_i. Sim honestly emulates the honest parties to participate in Π_{ACS}^{Q} and gets $\mathsf{CoreSet}$.
7. **Committing Local Inputs.** For each honest party $P_{j,i}$, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ to send a random message (of the same length) as $\mathsf{IS}^{P_{j,i}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,i}} \| \mathsf{mid})$ to $\mathcal{A}$. For each corrupted party $P_{j,i}$, Sim receives $\mathsf{IS}^{P_{j,i}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,i}} \| \mathsf{mid})$ from $\mathcal{A}$. Then:
   (a) For each $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharing generated in this phase by the parties in $\mathsf{CoreSet}$, Sim receives all the shares of corrupted members (of the additive sharings of virtual parties' shares) from the sets ($\mathsf{IS}^{P_{j,i}}$ of corrupted $P_{j,i}$) they committed. Since $\mathsf{IS}^{P_{j,i}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,i}} \| \mathsf{mid})$ and $\mathsf{seed}^{P_{j,i}}$ are known by Sim, Sim can compute $\mathsf{IS}^{P_{j,i}}$.
   (b) For each sharing generated by an honest party P_i, for each honest virtual server V_j, let the generated share of V_j be s (computed from $v_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ and k_{i,j_α} for $\alpha = 1, 2, 3$), and then Sim checks whether the committed input of each corrupted party $P_{j,\alpha}$'s shares of $\langle s \rangle$ (computed from $\mathsf{IS}^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\alpha}} \| \mathsf{mid})$ and $\mathsf{seed}^{P_{j,\alpha}}$) matches what Sim sends to him. If not, Sim sets $\mathsf{Corr}_j = 1$.
   (c) For each sharing generated by an corrupted party P_i, for each honest virtual server V_j, let the generated share of V_j be s (computed from $v_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ and k_{i,j_α} for $\alpha = 1, 2, 3$). Sim checks whether the committed value for each honest party $P_{j,\alpha}$'s shares of $\langle s \rangle$ (computed from $\mathsf{Sh}_i^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}_i^{P_{j,\alpha}} \| \mathsf{mid})$ and $\mathsf{seed}_i^{P_{j,\alpha}}$) matches what Sim receives from $\mathcal{A}$. If not, Sim sets $\mathsf{Corr}_j = 1$. In addition, Sim checks whether the committed shares from $\mathsf{Sh}_i^{P_{j,\alpha}}$ and $\mathsf{IS}^{P_{j,\alpha}}$ matches for each corrupted P_i and $P_{j,\alpha}$. If not, Sim sets $\mathsf{Corr}_j = 1$.
   (d) For the results of $\mathcal{F}_{\mathsf{ROT}}$, Sim checks whether for each honest virtual server V_j, the corrupted parties $P_{j,i}$ all committed their outputs correctly. For each honest virtual server V_j that fails in the check, Sim sets $\mathsf{ROT}_j = 1$.
   (e) For each corrupted party $P_{j,i}$, Sim retrieves $\Delta^{P_{j,i}}$ and $\lambda_w^{P_{j,i}}, k_{w,0}^{P_{j,i}}$ for each wire w that is not an output wire of an XOR gate of $\mathtt{Circ}^{V_j}$ from $\mathsf{IS}^{P_{j,i}}$.

Figure 25: The simulator for the sharing phase of Π.

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

Local Computation Phase

For each corrupted party $P_{j,i}$, Sim follows the protocol to do all the local computation of $P_{j,i}$ with their committed inputs.

Figure 26: The simulator for the local computing phase of Π.

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

Garbling Phase

For each virtual server V_j emulated by both honest and corrupted parties, Sim emulates each honest member $P_{j,i}$ and the random oracle $\mathcal{O}$ to communicate with corrupted members during the execution of Π_{in} for V_j. For each honest member $P_{j,i}$, Sim first waits until the $\mathsf{Dispersed}$ messages from all the instances of $\mathcal{F}_{\mathsf{AVID}}$ generated in the last step of the sharing phase whose dealer is another party $P_{j,\alpha}, \alpha \neq i$ are all delivered to $P_{j,i}$. Then, Sim begins to emulate $P_{j,i}$ in Π_{in}. For each round of Π_{in}, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ to send a random message (of the same length) as $\mathsf{MS}^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\alpha}} \| \mathsf{mid})$ to $\mathcal{A}$ Then, Sim waits to receive $\mathsf{MS}^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\alpha}} \| \mathsf{mid})$ from each corrupted member $P_{j,\alpha}$. If $\mathsf{MS}^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\alpha}} \| \mathsf{mid})$ for all $\alpha \neq i \in \{1, 2, 3\}$ are delivered to an honest party $P_{j,i}$, Sim begins the simulation of $P_{j,i}$ in the next round.

The execution of Π_{in} is simulated as follows:

1. **OT Extension.** Sim run the simulator of [KOS15] to interact with $\mathcal{A}$.
2. **Handling XOR gates.** For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wires a, b and output wire o, Sim follows the protocol to compute
$$k_{o,0}^{P_{j,i}} \| \lambda_o^{P_{j,i}} = (k_{a,0}^{P_{j,i}} \| \lambda_a^{P_{j,i}}) \oplus (k_{b,0}^{P_{j,i}} \| \lambda_b^{P_{j,i}})$$
gate by gate for each i such that $P_{j,i}$ is corrupted.
3. **Computing 1-Labels.** For each wire w in $\mathtt{Circ}^{V_j}$ that is not an output wire of an output gate. Sim follows the protocol to compute $k_{w,1}^{P_{j,i}} = k_{w,0}^{P_{j,i}} \oplus \Delta^{P_{j,i}}$ for each i such that $P_{j,i}$ is corrupted.
4. **Handling AND Gates.** For each AND gate g in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o:
   (a) For each execution of Π_{Mult} in this step, for each honest member $P_{j,\alpha}$ corrupted member $P_{j,\beta}$, Sim samples a random κ-bit string as $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$, samples a random bit as $y^{(\alpha)} \oplus b^{(\beta,\alpha)}$, and sends them to $P_{j,\beta}$ on behalf of $P_{j,\alpha}$. Then Sim receives $r_1^{(\beta,\alpha)} \oplus r_0^{(\beta,\alpha)} \oplus x^{(\beta)}$ and $y^{(\beta)} \oplus b^{(\alpha,\beta)}$ from $P_{j,\beta}$ and checks whether they are correctly computed with their committed inputs. If not, Sim sets $\mathsf{Comp}_j = 1$.
   (b) Sim honestly emulates the random oracle $\mathcal{O}$ and computes each corrupted member $P_{j,i}$'s shares of $\langle \boldsymbol{A}_{g,1}^{V_j} \rangle, \langle \boldsymbol{A}_{g,2}^{V_j} \rangle, \langle \boldsymbol{A}_{g,3}^{V_j} \rangle, \langle \boldsymbol{A}_{g,4}^{V_j} \rangle$ based on their committed inputs, the randomly sampled messages sent from honest members of V_j, and the outputs of $\mathcal{O}$.
5. **Handling Output Gates.** For each output gate with index k in $\mathtt{Circ}^{V_j}$ with input wire w:
   (a) For each execution of Π_{Mult} in this step, for each honest member $P_{j,\alpha}$ and each corrupted member $P_{j,\beta}$, Sim samples a random $(\kappa-1)$-bit string as $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$, samples a random bit as $y^{(\alpha)} \oplus b^{(\beta,\alpha)}$, and sends them to $P_{j,\beta}$ on behalf of $P_{j,\alpha}$. Then Sim receives $r_1^{(\beta,\alpha)} \oplus r_0^{(\beta,\alpha)} \oplus x^{(\beta)}$ and $y^{(\beta)} \oplus b^{(\alpha,\beta)}$ from $P_{j,\beta}$ and checks whether they are correctly computed with their committed inputs. If not, Sim sets $\mathsf{Comp}_j = 1$.
   (b) Sim honestly emulates the random oracle $\mathcal{O}$ and computes each corrupted member $P_{j,i}$'s shares of $\langle \mathbf{ct}_{w,0}^{V_j} \rangle, \langle \mathbf{ct}_{w,1}^{V_j} \rangle$ based on their committed inputs, the randomly sampled messages sent from honest members of V_j, and the outputs of $\mathcal{O}$.
6. **Masking Input Wire Values.** For each input wire w of $\mathtt{Circ}^{V_j}$:
   (a) Sim samples a random bit as each honest member $P_{j,\alpha}$'s share of $\langle x_w \oplus \lambda_w \rangle$ and sends it to each corrupted member $P_{j,\beta}$ on behalf of $P_{j,\alpha}$.
   (b) Sim receives each corrupted member $P_{j,\beta}$'s share of $\langle x_w \oplus \lambda_w \rangle$ and checks whether it matches the input he committed. If not, Sim sets $\mathsf{Comp}_j = 1$. Otherwise, Sim reconstructs $x_w \oplus \lambda_w$.
7. **Ouputting Output Masks.** For each input wire w of an output gate in $\mathtt{Circ}^{V_j}$, Sim computes corrupted members' shares of $\langle \lambda_w \rangle$ based on their committed inputs, the randomly sampled messages sent from honest members of V_j, and the outputs of $\mathcal{O}$.
8. **Encrypting Input Labels.** For each reconstruction of $\Sigma^{(2)}$-sharing in the evaluation phase of Π_0 whose receiver is V_j, we assume its the i-th reconstruction, and the sharing to be reconstructed is $[\boldsymbol{s}_i]^{(2)}$. Suppose the η-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_η in $\mathtt{Circ}^{V_j}$. For each of them and each honest member $P_{j,\alpha}$ of V_j, Sim computes
$$\mathsf{ct}_{j_\eta, s_{i,\eta}}^{(i,\alpha)} = \mathcal{O}(\boldsymbol{r}_{s_{i,\eta},\eta,\alpha} \| s_{i,\eta} \| i \| \alpha \| \eta \| j_\eta) \oplus \left(k_{w_{j_\eta}, v_{w_{j_\eta}} \oplus \lambda_{w_{j_\eta}}}^{P_{j,\alpha}} \| (v_{w_{j_\eta}} \oplus \lambda_{w_{j_\eta}})\right).$$
Then, Sim_2 samples a random κ-bit string as $\mathsf{ct}_{j_\eta, 1 \oplus s_{i,\eta}}^{(i,\alpha)}$.
9. **Outputting Garbled Circuits.** Sim computes the corrupted members' shares of $\langle \boldsymbol{A}_{g,1}^{V_j} \rangle, \langle \boldsymbol{A}_{g,2}^{V_j} \rangle, \langle \boldsymbol{A}_{g,3}^{V_j} \rangle, \langle \boldsymbol{A}_{g,4}^{V_j} \rangle$ for each AND gate and $\langle \mathbf{ct}_{w,0}^{V_j} \rangle, \langle \mathbf{ct}_{w,1}^{V_j} \rangle$ for each output wire w based on their committed inputs, the randomly sampled messages sent from honest members of V_j, and the outputs of $\mathcal{O}$.
10. After the execution of Π_{in} all the messages including the $\mathsf{Dispersed}$ messages of $\mathcal{F}_{\mathsf{AVID}}$ sent in the last round are delivered to each honest member $P_{j,i}$, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ to send a random message (of the same length) as $\mathsf{OS}^{P_{j,i}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,i}} \| \mathsf{mid})$ to $\mathcal{A}$. Besides, Sim receives $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ from each corrupted member $P_{j,\beta}$ and decrypts $\mathsf{OS}^{P_{j,\beta}}$ with $\mathsf{seed}_0^{P_{j,\beta}}$.

For each corrupted virtual server V_j, Sim honestly emulates the random oracle $\mathcal{O}$ to send outputs to corrupted members of V_j.

Figure 27: The simulator for the garbling phase of Π.

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

Verification Phase

1. **Verification of the Sharings.**

   (a) Sim emulates $\mathcal{F}_{\mathsf{Coin}}$ to receive $\mathsf{RandCoin}$ from the corrupted parties. When there are a honest parties that the $\mathsf{RandCoin}$ from them have been allowed to be delivered to $\mathcal{F}_{\mathsf{Coin}}$ and b corrupted parties that have sent $\mathsf{RandCoin}$ to $\mathcal{F}_{\mathsf{Coin}}$ such that $a+b \geq t+1$, Sim follows the protocol to sample $s \in \mathbb{F}_{2^{\kappa'}}$ randomly.

   (b) Sim emulates $\mathcal{F}_{\mathsf{Coin}}$ to send s to $\mathcal{A}$. If abort is received from $\mathcal{A}$, Sim emulates $\mathcal{F}_{\mathsf{Coin}}$ to send abort to $\mathcal{A}$ and aborts the protocol. After the simulation terminates, Sim outputs what $\mathcal{A}$ outputs. Let the pseudorandom coefficients for the $\Sigma^{(2)}$-sharings expanded from s be $(s_1, \ldots, s_{k_1}, s_1^{(2)}, \ldots, s_{k_2}^{(2)}, s_1^{(3)}, \ldots, s_{k_3}^{(3)}) \in \mathbb{F}_{2^{\kappa'}}^{k_1+k_2+k_3}$.

   (c) Let the set of the indices of Σ-sharings, random $\Sigma^{(2)}$-sharings, and random $\Sigma^{(3)}$-sharings (to be checked in this step) generated by corrupted parties be $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ respectively. Correspondingly, let the index set of sharings generated by honest parties be $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$. For these sharings generated by each corrupted party P_i, Sim computes the share s of each honest virtual server V_j based on each honest member $P_{j,\alpha}$'s share of $\langle s \rangle$ (computed from $v_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$and k_{i,j_α}) and each corrupted member $P_{j,\beta}$'s share of $\langle s \rangle$ (computed from $\mathsf{Sh}_i^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}_i^{P_{j,\alpha}} \| \mathsf{mid})$ and $\mathsf{seed}_i^{P_{j,\alpha}}$). Then, Sim computes $P_{j,1}, P_{j,2}, P_{j,3}$'s shares of

   $$\sum_{i \in \mathcal{C}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{C} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle,$$

   $$\sum_{i \in \mathcal{C}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{C} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle,$$

   and

   $$\sum_{i \in \mathcal{C}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{C} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle.$$

   (d) For each corrupted virtual server V_j, Sim follows the protocol to compute $P_{j,1}, P_{j,2}, P_{j,3}$'s shares of

   $$\sum_{i \in \mathcal{H}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle,$$

   $$\sum_{i \in \mathcal{H}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle,$$

   and

   $$\sum_{i \in \mathcal{H}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle.$$

   Also, for each honest virtual server V_j, Sim follows the protocol to compute each corrupted member $P_{j,\alpha}$'s shares of them. Then, Sim randomly samples the secret of the sharing for each honest virtual server V_j based on the fact that all the virtual servers' secrets form a $\Sigma_{\kappa'}$-sharing, a $\Sigma_{\kappa'}^{(2)}$-sharing, and a $\Sigma_{\kappa'}^{(3)}$-sharing.

   (e) For each honest virtual server V_j, Sim computes $P_{j,1}, P_{j,2}, P_{j,3}$'s shares of

   $$\sum_{i=1}^{k_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{P_i \in \mathsf{CoreSet}} \langle \boldsymbol{r}^{(i,V_j)} \rangle = \sum_{i \in \mathcal{C}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{C} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle + \sum_{i \in \mathcal{H}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle,$$

$$\sum_{i=1}^{k_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{P_i \in \mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle = \sum_{i \in \mathcal{C}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{C} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle + \sum_{i \in \mathcal{H}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle,$$

and

$$\sum_{i=1}^{k_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j} \rangle + \sum_{P_i \in \mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle = \sum_{i \in \mathcal{C}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{C} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle + \sum_{i \in \mathcal{H}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle.$$

Sim then performs a similar computation on the sharings of all-zero secrets as in Step 1.(a)-(e). Then, Sim follows the protocol to compute $\mathsf{VS}^{P_{j,i}} \oplus \mathcal{O}(\mathsf{seed}^{P'_{j,i}} \| \mathsf{mid})$ and sends it to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{AVID}}$ for each honest member $P_{j,i}$ of V_j. Sim then emulates $\mathcal{F}_{\mathsf{AVID}}$ to receive $\mathsf{VS}^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}^{P'_{j,\alpha}} \| \mathsf{mid})$ from each corrupted member $P_{j,\alpha}$ of V_j. Sim computes $\mathsf{VS}^{P_{j,\alpha}}$ for each corrupted member $P_{j,\alpha}$ of each honest virtual server V_j from $\mathsf{VS}^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}^{P'_{j,\alpha}} \| \mathsf{mid})$ and $\mathsf{seed}^{P'_{j,\alpha}}$ and checks whether the set is correctly computed from $\mathsf{Sh}_i^{P_{j,\alpha}}$ for $P_i \in \mathsf{CoreSet}$. If not, Sim sets $\mathsf{Comp}_j = 1$.

2. **Verification of Local Computation.**
   (a) For each $j = 1, \ldots, N$, upon the $\mathsf{Dispersed}$ messages for all the instances of $\mathcal{F}_{\mathsf{AVID}}$ whose dealer is $P_{j,\alpha}, \alpha = 1, 2, 3$ called in during the execution of Π_{in} for V_j and Step 1 of the verification phase are delivered to an honest party P_i, Sim regarded that P_i has sent 1 to $\mathcal{F}_{\mathsf{ABA}}^{(j)}$. Sim emulates $\mathcal{F}_{\mathsf{ABA}}^{(j)}$ to receive inputs of corrupted parties from $\mathcal{A}$ and faithfully emulates $\mathcal{F}_{\mathsf{ABA}}$ to send outputs to $\mathcal{A}$. If for $N/4$ $j \in \{1, \ldots, N\}$, the output of $\mathcal{F}_{\mathsf{ABA}}$ is 1 and has been delivered to an honest party P_i, Sim regard that P_i has sent 1 to $\mathcal{F}_{\mathsf{ABA}}^{(\alpha)}$ for each $\mathcal{F}_{\mathsf{ABA}}^{(\alpha)}$ that has not received input from P_i.
   (b) Upon the emulation of all $\mathcal{F}_{\mathsf{ABA}}^{(j)}, j = 1, \ldots, N$ terminates, Sim follows the protocol to obtain the set Ter.
   (c) For each $V_j \in \mathsf{Ter}$, Sim faithfully emulates $\mathcal{F}_{\mathsf{Coin}}$ to sample and send the output coin to $\mathcal{A}$. If abort is received from $\mathcal{A}$, Sim emulates $\mathcal{F}_{\mathsf{Coin}}$ to send abort to $\mathcal{A}$ and aborts the protocol. Then, Sim obtains the set of checked parties in Ter. For each checked honest virtual server V_j, Sim sets $\mathsf{Check}_j = 1$. If there are more than $N/312$ virtual servers V_j with $\mathsf{Check}_j = 1$, Sim aborts the simulation.
   (d) If for over $N/528$ $V_j \in \mathsf{Ver}$ it holds that $\mathsf{Comp}_j = 1$, Sim sets $\mathsf{CompCheck} = 1$. If for over $N/528$ $V_j \in \mathsf{Ver}$ it holds that $\mathsf{Corr}_j = 1$, Sim sets $\mathsf{CorrCheck} = 1$. If for over $N/220$ $V_j \in \mathsf{Ver}$ it holds that $\mathsf{ROT}_j = 1$, Sim sets $\mathsf{OTCheck} = 1$.
   (e) For each honest member $P_{j,\alpha}$ of each honest virtual server $V_j \in \mathsf{Ter}$ with $\mathsf{Check}_j = 1$:
      i. Sim randomly samples $P_{j,\alpha}$'s shares of all the sharings generated by an honest party in the sharing phase based on corrupted parties' shares except the additive sharings for V_j's shares of $[\boldsymbol{r}_1^{(i')}]_{\kappa'}, [\boldsymbol{r}_2^{(i')}]_{\kappa'}^{(2)}, [\boldsymbol{r}_3^{(i')}]_{\kappa'}^{(2)}$ generated by the last honest party $P_{i'}, i' \in \mathcal{H} \cap \mathsf{CoreSet}$. For $P_{j,\alpha}$'s shares for the remaining 3 sharings, Sim computes them based on his shares of $\sum_{i \in \mathcal{H}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle, \sum_{i \in \mathcal{H}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle, \sum_{i \in \mathcal{H}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j} \rangle + \sum_{i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle$ and $\sum_{i \in \mathcal{H}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{i \in \mathcal{H} \cap \mathsf{CoreSet} \setminus \{i'\}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle, \sum_{i \in \mathcal{H}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{i \in \mathcal{H} \cap \mathsf{CoreSet} \setminus \{i'\}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle, \sum_{i \in \mathcal{H}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j} \rangle + \sum_{i \in \mathcal{H} \cap \mathsf{CoreSet} \setminus \{i'\}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle$. Similarly for the sharings $[\boldsymbol{o}_1^{(i')}]_{\kappa'}, [\boldsymbol{o}_2^{(i')}]_{\kappa'}^{(2)}, [\boldsymbol{o}_3^{(i')}]_{\kappa'}^{(2)}$.
      ii. Sim randomly samples a $(\kappa - 1)$-bit string as $\Delta^{P_{j,\alpha}}$.
      iii. For each w that is not an output wire of an XOR gate or output gate in $\mathtt{Circ}^{V_j}$ nor an input wire of $\mathtt{Circ}^{V_j}$, Sim randomly samples $\lambda_w^{P_{j,\alpha}}, k_{w,0}^{P_{j,\alpha}}$.
      iv. For each input wire w of $\mathtt{Circ}^{V_j}$, Sim computes $P_{j,\alpha}$'s share $\lambda_w^{P_{j,\alpha}}$ of $\langle \lambda_w \rangle$ based on his shares of $\langle x_w \oplus \lambda_w \rangle$ (which has been generated in the garbling phase) and $\langle x_w \rangle$ (which has been generated in Step 2.(d).i.).

v. For each XOR gate in $\texttt{Circ}^{V_j}$ with input wires a, b and output wire o, Sim computes

$$k_{o,0}^{P_{j,\alpha}} \| \lambda_o^{P_{j,\alpha}} = (k_{a,0}^{P_{j,\alpha}} \| \lambda_a^{P_{j,\alpha}}) \oplus (k_{b,0}^{P_{j,\alpha}} \| \lambda_b^{P_{j,\alpha}}).$$

This computation is performed gate by gate.

vi. For each wire w in $\texttt{Circ}^{V_j}$ that is not an output wire of an output gate, Sim computes $k_{w,1}^{P_{j,\alpha}} = k_{w,0}^{P_{j,\alpha}} \oplus \Delta^{P_{j,\alpha}}$.

vii. For each execution of Π_{Mult} in the garbling phase, for each corrupted member $P_{j,\beta}$ of V_j, Sim already has $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$, $(b^{(\alpha,\beta)}, r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)})$, $y^{(\alpha)} \oplus r_{b^{(\beta,\alpha)}}^{(\beta,\alpha)}$, and $x^{(\alpha)}, y^{(\alpha)}$. With these values, Sim computes $P_{j,\alpha}$'s output of each ROT instance.

viii. For each AND gate in $\texttt{Circ}^{V_j}$ with input wire a, b and output wire o, Sim computes honest members' shares of $\langle A_{g,2i_0+i_1}^{P_{j,i}} \rangle$ for each $(i_0, i_1) = (0,0), (0,1), (1,0), (1,1)$ and $i = 1, 2, 3$ by following the computation process in the Π_{in}.

ix. For each output gate in $\texttt{Circ}^{V_j}$ with input wire w, Sim computes honest members' shares of $\langle \mathsf{ct}_{w,i_2}^{P_{j,i}} \rangle$ for each $i_2 = 0, 1$ and $i = 1, 2, 3$ by following Π_{in}. Sim knows all the elements in $\mathsf{IS}^{P_{j,\alpha}}, \mathsf{Sh}_i^{P_{j,\alpha}}$ for each honest party P_i after completing the above steps.

(f) For each checked virtual server V_j:

i. Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ (that commits $\mathsf{VS}^{V_{j,\alpha}}, \alpha = 1, 2, 3$) to receive $(\mathsf{Retrieve}, P_i)$ for each party P_i. For each honest party, Sim regards $(\mathsf{Retrieve}, P_i)$ to $\mathcal{F}_{\mathsf{AVID}}$ upon the random coins generated by $\mathcal{F}_{\mathsf{Coin}}$ are delivered. For corrupted party P_i, if $t+1$ parties' messages $(\mathsf{Retrieve}, P_i)$ have been delivered to $\mathcal{F}_{\mathsf{AVID}}$, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ to send $\mathsf{VS}^{V_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}^{P'_{j,\alpha}}), \alpha = 1, 2, 3$ to P_i.

ii. Sim randomly samples $q^{P'_{j,i}}$ based on $\{q^{P'_{j,i}}(\alpha_u)\}_{u \in \mathcal{C}}$ and follows the protocol to emulate honest parties to participate in the OEC process for reconstructing the seed $\mathsf{seed}^{P'_{j,i}}$ for each $i = 1, 2, 3$.

iii. Sim follows the protocol to check whether $[\tau]_{\times\kappa'}, [\tau_2]_{\times\kappa'}^{(2)}, [\tau_3]_{\times\kappa'}^{(3)}$ (where corrupted parties' shares are obtained from $\mathsf{VS}^{P_{j,i}}$ for corrupted $P_{j,i}$) are valid shares for $\Sigma_{\times\kappa'}, \Sigma_{\times\kappa'}^{(2)}, \Sigma_{\times\kappa'}^{(3)}$-sharings and whether $[\tau']_{\times\kappa'}, [\tau'_2]_{\times\kappa'}^{(2)}, [\tau'_3]_{\times\kappa'}^{(3)}$ are valid shares for $\Sigma_{\times\kappa'}, \Sigma_{\times\kappa'}^{(2)}, \Sigma_{\times\kappa'}^{(3)}$-sharings with all-zero secrets. If not, Sim aborts the protocol on behalf of each honest party P_i. After completing the simulation, Sim outputs what $\mathcal{A}$ outputs.

iv. Sim randomly samples $\mathsf{seed}_\alpha^{P_{j,\beta}}, \mathsf{seed}^{P_{j,\beta}}, \mathsf{seed}_0^{P_{j,\beta}}$ for each honest party P_α and honest party $P_{j,\beta}$. Then, Sim samples $q_\alpha^{P_{j,\beta}}(\alpha_u), q^{P_{j,\beta}}(\alpha_u), q_0^{P_{j,\beta}}(\alpha_u)$ for honest parties P_u based on $\mathsf{seed}_\alpha^{P_{j,\beta}}, \mathsf{seed}^{P_{j,\beta}}, \mathsf{seed}_0^{P_{j,\beta}}$ and $q_\alpha^{P_{j,\beta}}(\alpha_v), q^{P_{j,\beta}}(\alpha_v), q_0^{P_{j,\beta}}(\alpha_v)$ for corrupted parties P_v. While emulating the random oracle $\mathcal{O}$, Sim uses $\mathsf{Sh}_\alpha^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$ and $\mathsf{Sh}_\alpha^{P_{j,\beta}}$ to compute the corresponding output $\mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$, uses $\mathsf{IS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$ and $\mathsf{IS}^{P_{j,\beta}}$ to compute the corresponding output $\mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, uses $\mathsf{MS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$ and $\mathsf{MS}^{P_{j,\beta}}$ to compute the corresponding output $\mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, and uses $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ and $\mathsf{OS}^{P_{j,\beta}}$ to compute the corresponding output $\mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$. If some of these queries to the random oracle have been queried by $\mathcal{A}$, Sim aborts the simulation.

v. Sim follows the protocol to emulate honest parties to participate in the OEC process of reconstructing the seeds $\mathsf{seed}_\alpha^{P_{j,\beta}}, \mathsf{seed}^{P_{j,\beta}}, \mathsf{seed}_0^{P_{j,\beta}}$ for each $\alpha = 1, \ldots, n$ and $\beta = 1, 2, 3$.

vi. Sim follows the protocol to emulate each honest party P_i to check whether the local computation of V_j is performed correctly and whether the committed outputs from $\mathcal{F}_{\mathsf{ROT}}$ are valid. If not, Sim aborts the protocol on behalf of P_i. After completing the simulation, Sim outputs what $\mathcal{A}$ outputs.

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

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

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

(j) Sim chooses a subset $\mathcal{H}_{\mathsf{vir}} \subset \mathsf{Ter}$ of $|\mathsf{Ter} \cup \mathsf{Corr}| - N/20 \geq N/5$ honest virtual servers (here Corr is the set of corrupted virtual servers), where each $V_j \in \mathcal{H}_{\mathsf{vir}}$ satisfies $\mathsf{Corr}_j = \mathsf{Comp}_j = \mathsf{Check}_j = \mathsf{ROT}_j = 0$. Then, Sim checks whether the shares of the virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of the $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharings generated by corrupted parties in $\mathsf{CoreSet}$ that are checked in Step 1 of the verification phase (computed from $\mathsf{Sh}_i^{P_{j,\alpha}}, j = 1, \ldots, N, \alpha = 1, 2, 3$ for each corrupted generator P_i) are valid, If not, Sim aborts the

simulation. Otherwise, Sim chooses a valid sharing as each sharing generated by the corrupted party that matches the shares for $V_j \in \mathcal{H}_{\mathsf{vir}}$ and then reconstructs the secret.

(k) For each honest virtual server $V_j \in \mathsf{Ter}\backslash\mathcal{H}_{\mathsf{vir}}$ with $\mathsf{Check}_j = 0$:

i. Sim randomly samples $P_{j,\alpha}$'s shares of all the sharings generated by an honest party in the sharing phase based on corrupted parties' shares except the additive sharings for V_j's shares of $[\boldsymbol{r}_1^{(i')}]_{\kappa'}, [\boldsymbol{r}_2^{(i')}]_{\kappa'}^{(2)}, [\boldsymbol{r}_3^{(i')}]_{\kappa'}^{(2)}$ generated by the last honest party $P_{i'}, i' \in \mathcal{H} \cap \mathsf{CoreSet}$. For $P_{j,\alpha}$'s shares for the remaining 3 sharings, Sim computes them based on his shares of $\sum_{i\in\mathcal{H}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j}\rangle + \sum_{i\in\mathcal{H}\cap\mathsf{CoreSet}} \langle \boldsymbol{r}_1^{(i,V_j)}\rangle, \sum_{i\in\mathcal{H}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j}\rangle + \sum_{i\in\mathcal{H}\cap\mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_j)}\rangle, \sum_{i\in\mathcal{H}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j}\rangle + \sum_{i\in\mathcal{H}\cap\mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_j)}\rangle$ and $\sum_{i\in\mathcal{H}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j}\rangle + \sum_{i\in\mathcal{H}\cap\mathsf{CoreSet}\backslash\{i'\}} \langle \boldsymbol{r}_1^{(i,V_j)}\rangle, \sum_{i\in\mathcal{H}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j}\rangle + \sum_{i\in\mathcal{H}\cap\mathsf{CoreSet}\backslash\{i'\}} \langle \boldsymbol{r}_2^{(i,V_j)}\rangle, \sum_{i\in\mathcal{H}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j}\rangle + \sum_{i\in\mathcal{H}\cap\mathsf{CoreSet}\backslash\{i'\}} \langle \boldsymbol{r}_3^{(i,V_j)}\rangle$. Similarly for the sharings $[\boldsymbol{o}_1^{(i')}]_{\kappa'}, [\boldsymbol{o}_2^{(i')}]_{\kappa'}^{(2)}, [\boldsymbol{o}_3^{(i')}]_{\kappa'}^{(2)}$.

ii. Sim randomly samples a $(\kappa-1)$-bit string as $\Delta^{P_{j,\alpha}}$.

iii. For each w that is not an output wire of an XOR gate or output gate in $\mathtt{Circ}^{V_j}$ nor an input wire of $\mathtt{Circ}^{V_j}$, Sim randomly samples $\lambda_w^{P_{j,\alpha}}, k_{w,0}^{P_{j,\alpha}}$.

iv. For each input wire w of $\mathtt{Circ}^{V_j}$, Sim computes $P_{j,\alpha}$'s share $\lambda_w^{P_{j,\alpha}}$ of $\langle\lambda_w\rangle$ based on his shares of $\langle x_w \oplus \lambda_w\rangle$ (which has been generated in the garbling phase) and $\langle x_w\rangle$ (which has been generated in Step 2.(k).i.).

v. For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wires a, b and output wire o, Sim computes

$$k_{o,0}^{P_{j,\alpha}} \| \lambda_o^{P_{j,\alpha}} = (k_{a,0}^{P_{j,\alpha}} \| \lambda_a^{P_{j,\alpha}}) \oplus (k_{b,0}^{P_{j,\alpha}} \| \lambda_b^{P_{j,\alpha}}).$$

This computation is performed gate by gate.

vi. For each wire w in $\mathtt{Circ}^{V_j}$ that is not an output wire of an output gate, Sim computes $k_{w,1}^{P_{j,\alpha}} = k_{w,0}^{P_{j,\alpha}} \oplus \Delta^{P_{j,\alpha}}$.

vii. For each execution of Π_{Mult} in the garbling phase, for each corrupted member $P_{j,\beta}$ of V_j, Sim already has $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$, $(b^{(\alpha,\beta)}, r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)})$, $y^{(\alpha)} \oplus r_{b^{(\beta,\alpha)}}^{(\beta,\alpha)}$, and $x^{(\alpha)}, y^{(\alpha)}$. With these values, Sim computes $P_{j,\alpha}$'s output of each ROT instance.

viii. For each AND gate in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, Sim computes honest members' shares of $\langle A_{g,2i_0+i_1}^{P_{j,i}}\rangle$ for each $(i_0, i_1) = (0,0), (0,1), (1,0), (1,1)$ and $i = 1, 2, 3$ by following the computation process in the Π_{in}.

(l) Sim randomly samples $\mathsf{seed}_0^{P_{j,\alpha}}$ for each honest member $P_{j,\alpha}$ of each unchecked virtual server $V_j \in \mathsf{Ter}$. Then, Sim samples $q_0^{P_{j,\beta}}(\alpha_u)$ for honest parties P_u based on $\mathsf{seed}_0^{P_{j,\beta}}$ and $q_0^{P_{j,\beta}}(\alpha_v)$ for corrupted parties P_v.

(m) Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ that commits each $\mathsf{OS}^{P_{j,\alpha}}, P_j \in \mathsf{Ter}, \alpha = 1, 2, 3$ to receive $(\mathsf{Retrieve}, P_i)$ for all $i = 1, \ldots, n$ from corrupted parties. Sim regards that an honest party has sent $(\mathsf{Retrieve}, P_i)$ when the check of the honest party passes. When $t+1$ parties have sent $(\mathsf{Retrieve}, P_i)$ to $\mathcal{F}_{\mathsf{AVID}}$ for a corrupted party P_i, Sim emulates $\mathcal{F}_{\mathsf{AVID}}$ to send $\mathsf{OS}^{P_{j,\alpha}} \oplus \mathsf{seed}_0^{P_{j,\alpha}}$ to P_i. Then, Sim honestly emulates honest parties to run the OEC process of reconstructing $\mathsf{seed}_0^{P_{j,\alpha}}$ to each party. Sim then uses $\mathsf{OS}^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\alpha}} \| \mathsf{mid})$ and $\mathsf{OS}^{P_{j,\alpha}}$ to compute the corresponding output $\mathcal{O}(\mathsf{seed}_0^{P_{j,\alpha}} \| \mathsf{mid})$ for each honest member $P_{j,\alpha}$ of V_j. If some of these queries to $\mathcal{O}$ have been queried by the adversary before the OEC process of the seeds, Sim aborts the simulation.

3. Sim sends $\mathsf{CoreSet}$ to $\mathcal{F}$. Then, Sim reconstructs the secrets of input sharings generated by each corrupted party $P_i \in \mathsf{CoreSet}$. Sim then retrieves P_i's input from these secrets and sends it to $\mathcal{F}$.

Figure 28: The simulator for the verification phase of Π.

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

Evaluation Phase

1. **Reconstructing the Garbled Circuit.** For each $i = 1, \ldots, \mathsf{rec}$, Sim computes the shares of $[\boldsymbol{s}_i]^{(2)}$ for virtual servers of $\mathcal{C}_{\mathsf{Vir}} = \mathsf{Corr} \cup \mathsf{Ter}\backslash\mathcal{H}_{\mathsf{vir}}$ by following Π_0 (where input sharings are from parties in $\mathsf{CoreSet}$,

the preprocessing datas are also prepared in previous steps). Then, Sim randomly samples the shares of $[\boldsymbol{s}_i]^{(2)}$ for virtual servers of $\mathcal{H}_{\mathsf{vir}}$ based on the shares for virtual servers of $\mathcal{C}_{\mathsf{vir}}$. Then, Sim does the following:

(a) If the receiver R_i of $[\boldsymbol{s}_i]$ is an honest client, Sim samples a random k-bit string as $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$ for each $\beta = 1,2,3$ and $\alpha = 1,\ldots,\kappa$. Then, for $\alpha = 1,\ldots,\kappa$, Sim randomly samples the shares for virtual servers in $\mathcal{H}_{\mathsf{Vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}]^{(3)},\ldots,[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,c}]^{(3)}$ based on the shares for the virtual servers in $\mathcal{C}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$ for each $\beta = 1,2,3$.

(b) If the receiver R_i of the i-th reconstruction is a virtual server in $\mathcal{H}_{\mathsf{vir}}$, Sim samples a random κ-bit string as $\boldsymbol{r}_{s_{i,\eta},\eta,1} = (r^{(1)}_{s_{i,\eta},\eta,1},\ldots,r^{(\kappa)}_{s_{i,\eta},\eta,1})$ for each $\eta = 1,\ldots,k$, where $s_{i,\eta}$ is the η-th bit of $\boldsymbol{s}_i$. Then, for $\alpha = 1,\ldots,\kappa$, let $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1} = (r^{(\alpha)}_{s_{i,1},1,1},r^{(\alpha)}_{s_{i,2},2,1},\ldots,r^{(\alpha)}_{s_{i,k},k,1})$, Sim randomly samples the shares for virtual servers in $\mathcal{H}_{\mathsf{Vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}]^{(3)}$ based on the shares of the virtual servers in $\mathcal{C}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},1}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},1} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},1}]$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}$.

(c) For each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$ and each $a = 1,\ldots,\ell^2$, let $s^{V_j}_{i,a}$ be the a-th bit of V_j's share of $[\boldsymbol{s}_i]$. Sim computes $Y^{P_{j,\beta}}_{(i-1)\ell^2+a,s^{V_j}_{i,a}}$ for each $\beta = 1,2,3$ with the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}]^{(3)}$ whose secret is held by the honest parties (emulating clients and honest members of virtual servers in $\mathcal{H}_{\mathsf{vir}}$) and each pair of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ whose secrets is held by other parties, where the computational process of each $Y^{P_{j,\beta}}_{(i-1)\ell^2+a,s^{V_j}_{i,a}}$ is the same as in the protocol with each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ (whose secret is held by an honest party) being regarded as $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$.

For each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$:

(a) For each wire w in $\mathtt{Circ}^{V_j}$ that is not an input wire of the circuit and is not an output wire of an XOR gate or an output gate, Sim samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string $k^{P_{j,\alpha}}_{w,v_w\oplus\lambda_w}$ for each honest member $P_{j,\alpha}$ of V_j. For each input wire of $\mathtt{Circ}^{V_j}$, Sim samples a random $(\kappa-1)$-bit string $k^{P_{j,\alpha}}_{w,v_w\oplus\lambda_w}$.

(b) For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wires a,b and output wire o, Sim computes $k^{P_{j,\alpha}}_{o,v_o\oplus\lambda_o} = k^{P_{j,\alpha}}_{a,v_a\oplus\lambda_a} \oplus k^{P_{j,\alpha}}_{b,v_b\oplus\lambda_b}$ and $v_o \oplus \lambda_o = (v_a \oplus \lambda_a) \oplus (v_b \oplus \lambda_b)$ gate by gate from the first layer.

(c) For each AND gate g in $\mathtt{Circ}^{V_j}$ with input wire a,b and output wire o, Sim computes the ciphertext encrypted by $\{k^{P_{j,i}}_{a,v_a\oplus\lambda_a}, k^{P_{j,i}}_{b,v_b\oplus\lambda_b}\}^3_{i=1}$ by:

$$\left(\bigoplus_{i=1}^{3}\left(\mathcal{O}(k^{P_{j,i}}_{a,v_a\oplus\lambda_a}\|(v_a\oplus\lambda_a)\|k^{P_{j,i}}_{b,v_b\oplus\lambda_b}\|(v_b\oplus\lambda_b)\|i\|j\|\beta\|g)\right)\right) \oplus k^{P_{j,\beta}}_{o,v_o\oplus\lambda_o}\|(v_o\oplus\lambda_o).$$

Then Sim samples 3 random κ-bit strings as the other 3 ciphertexts for this gate g and each $\beta = 1,2,3$.

(d) For each input wire w of an output gate in $\mathtt{Circ}^{V_j}$, the output gate outputs a bit of V_j's share of a $\Sigma^{(2)}$-sharing that needs reconstruction in Π_0 which has already be computed. Sim sets the output wire value v_w to be the corresponding bit. Then, Sim computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$.

(e) For each output gate in $\mathtt{Circ}^{V_j}$ indexed $\mathsf{k} = 1,\ldots,\ell^2\mathsf{rec}$ with input wire w, Sim computes

$$\mathsf{ct}^{P_{j,\beta}}_{w,v_w\oplus\lambda_w} = \left(\bigoplus_{i=1}^{3}\mathcal{O}(k^{P_{j,i}}_{w,v_w\oplus\lambda_w}\|(v_w\oplus\lambda_w)\|i\|j\|\beta\|w)\right) \oplus Y^{P_{j,\beta}}_{\mathsf{k},\lambda_w}.$$

Then Sim samples a random $c\ell\kappa$-bit string as the other ciphertext for this wire w and each $\beta = 1,2,3$.

Now, Sim has already got the garbled circuits (including input labels, output masks, and the ciphertexts for AND gates and output gates) of all $V_j \in \mathsf{Ter}$, i.e. $\mathsf{OS}^{P_{j,\alpha}}, P_j \in \mathsf{Ter}, \alpha = 1,2,3$. Then, while emulating the random oracle $\mathcal{O}$, Sim uses $\mathsf{OS}^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\alpha}}_0\|\mathsf{mid})$ and $\mathsf{OS}^{P_{j,\alpha}}$ to compute the corresponding output $\mathcal{O}(\mathsf{seed}^{P_{j,\alpha}}_0\|\mathsf{mid})$.

2. **Evaluating the Circuit.** Sim follows the protocol to emulate honest parties to evaluate the circuit and compute the output.
3. Sim outputs what $\mathcal{A}$ outputs.

Figure 29: The simulator for the evaluation phase of Π.

We construct the following hybrids:

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

$\mathbf{Hyb}_1$: In this hybrid, while determining the virtual servers, Sim aborts the simulation if over $N/26$ virtual servers are corrupted. By Chernoff bound, if the virtual servers are truly randomly determined, the probability that over $N/26$ virtual servers are corrupted is negligible. Thus, the distributions of $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ are statistically close.

$\mathbf{Hyb}_2$: In this hybrid, whenever an honest party (either a client or a server) generates a random $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharing (including the interleaved secret sharings of them) for the virtual servers, Sim first generates the corrupted virtual servers' shares, and then randomly samples the honest virtual servers' shares based on corrupted virtual servers' shares and the secret. Since for all these sharings, the set of corrupted virtual servers' shares is independent of the secret, we only change the order of generating the honest and corrupted virtual servers' shares of the sharings. This doesn't change the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

$\mathbf{Hyb}_3$: In this hybrid, whenever an honest party generates an additive sharing for the parties who act as a virtual server, Sim first generates the corrupted parties' shares, and then randomly samples the honest parties' shares based on corrupted parties' shares and the secret. Since for all these sharings, the set of corrupted parties' shares is independent of the secret, we only change the order of generating the honest and corrupted parties' shares of the sharings. This doesn't change the output distribution. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same output distribution.

$\mathbf{Hyb}_4$: In this hybrid, Sim additionally sets $\mathsf{CompCheck} = \mathsf{CorrCheck} = \mathsf{OTCheck} = 0$ at the end of the setup phase. Then, for each honest virtual server V_j, Sim sets $\mathsf{Corr}_j = \mathsf{Comp}_j = \mathsf{ROT}_j = \mathsf{Check}_j = 0$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ have the same output distribution.

$\mathbf{Hyb}_5$: In this hybrid, during the sharing phase, whenever Sim generates a share $\boldsymbol{v}_{i,\alpha}^{V_j}$ for an honest $P_{j,\alpha} = P_{j_\alpha}$ via AVID on behalf of an honest party P_i, Sim first samples $\boldsymbol{v}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ randomly and then computes $\mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid}) = \boldsymbol{v}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid}) \oplus \boldsymbol{v}_{i,\alpha}^{V_j}$. Since $\boldsymbol{v}_{i,\alpha}^{V_j}$ is uniformly random when $k_{i,j_\alpha} \| \mathsf{mid}$ is not queried by $\mathcal{A}$ for all these mid. Similar for $\boldsymbol{v}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$. We only change the order of generating $\boldsymbol{v}_{i,\alpha}^{V_j}$ and $\boldsymbol{v}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ without changing their distributions if $k_{i,j_\alpha} \| \mathsf{mid}$ is not queried by $\mathcal{A}$. Therefore, the distribution only changes when some $k_{i,j_\alpha} \| \mathsf{mid}$ is queried by $\mathcal{A}$. Since k_{i,j_α} is sampled randomly in $\mathbb{F}_{2^\kappa}$ and it's not used in sending any message to $\mathcal{A}$ in this hybrid, the probability that $\mathcal{A}$ queries some $k_{i,j_\alpha} \| \mathsf{mid}$ during the execution is $\mathsf{poly}(\kappa)/2^\kappa$, which is negligible. Thus, the distributions of $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ are computationally indistinguishable.

$\mathbf{Hyb}_6$: In this hybrid, during the sharing phase, after $\mathsf{IS}^{P_{j,i}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,i}} \| \mathsf{mid})$ is received from a corrupted party $P_{j,i}$, Sim computes $\mathsf{IS}^{P_{j,i}}$ from $\mathsf{IS}^{P_{j,i}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,i}} \| \mathsf{mid})$ and $\mathsf{seed}^{P_{j,i}}$. Then, Sim additionally performs the following checks:

1. For each sharing generated by an honest party P_i, for each honest virtual server V_j, let the generated share of V_j be s (computed from $v_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ and k_{i,j_α} for $\alpha = 1, 2, 3$), and then Sim checks whether the committed input of each corrupted party $P_{j,\alpha}$'s shares of $\langle s \rangle$ (computed from $\mathsf{IS}^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\alpha}} \| \mathsf{mid})$ and $\mathsf{seed}^{P_{j,\alpha}}$) matches what Sim sends to him. If not, Sim sets $\mathsf{Corr}_j = 1$.

2. For each sharing generated by an corrupted party P_i, for each honest virtual server V_j, let the generated share of V_j be s (computed from $v_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ for $\alpha = 1, 2, 3$), and then Sim checks whether the committed share of each honest party $P_{j,\alpha}$'s shares of $\langle s \rangle$ (computed from $\mathsf{Sh}_i^{P_{j,\alpha}} \oplus \mathcal{O}(\mathsf{seed}_i^{P_{j,\alpha}} \| \mathsf{mid})$ and $\mathsf{seed}_i^{P_{j,\alpha}}$) matches what Sim receives from $\mathcal{A}$. If not, Sim sets $\mathsf{Corr}_j = 1$. In addition, Sim checks

whether the committed shares from $\mathsf{Sh}_i^{P_{j,\alpha}}$ and $\mathsf{IS}^{P_{j,\alpha}}$ matches for each corrupted P_i and $P_{j,\alpha}$. If not, Sim sets $\mathsf{Corr}_j = 1$.

3. For the results of $\mathcal{F}_{\mathsf{ROT}}$, Sim checks whether for each honest virtual server V_j, the corrupted parties $P_{j,i}$ all committed their outputs correctly. For each honest virtual server V_j that fails in the check, Sim sets $\mathsf{ROT}_j = 1$.

This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ have the same output distribution.

$\mathbf{Hyb}_7$: In this hybrid, during the garbling phase, for each honest virtual server V_j, for each execution of Π_{Mult}, Sim doesn't follow the protocol to compute $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$ and $y^{(\alpha)} \oplus b^{(\beta,\alpha)}$ for each honest member $P_{j,\alpha}$ and corrupted member $P_{j,\beta}$. Instead, Sim randomly samples a $(\kappa-1)$-bit string as $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$ and a random bit as $y^{(\alpha)} \oplus b^{(\beta,\alpha)}$. Then, Sim computes $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)}$ based on $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$ and $x^{(\alpha)}$ and then determines $(r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)})$ based on $(b^{(\alpha,\beta)}, r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)})$. Similarly, Sim computes $b^{(\beta,\alpha)}$ based on $y^{(\alpha)} \oplus b^{(\beta,\alpha)}$ and $y^{(\alpha)}$ to determine $r_{b^{(\beta,\alpha)}}^{(\beta,\alpha)}$. Since $(r_0^{(\alpha,\beta)}, r_1^{(\alpha\beta)})$ are sampled randomly based on $(b^{(\alpha,\beta)}, r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)})$, $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$ is a random $(\kappa-1)$-bit string. Since $b^{(\beta,\alpha)}$ is a random bit, so is $y^{(\alpha)} \oplus b^{(\beta,\alpha)}$. Thus, we only change the order of generating $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$ and $(r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)})$, $y^{(\alpha)} \oplus b^{(\beta,\alpha)}$ and $b^{(\beta,\alpha)}$ without changing their distributions. Thus, $\mathbf{Hyb}_7$ and $\mathbf{Hyb}_6$ have the same output distribution.

$\mathbf{Hyb}_8$: In this hybrid, during the garbling phase, for each execution of Π_{Mult} executed by members emulating an honest virtual server V_j. Sim additionally checks whether $r_1^{(\beta,\alpha)} \oplus r_0^{(\beta,\alpha)} \oplus x^{(\beta)}$ and $y^{(\beta)} \oplus b^{(\alpha,\beta)}$ from $P_{j,\beta}$ for each honest member $P_{j,\alpha}$ and corrupted member $P_{j,\beta}$ are correctly computed with corrupted members' committed inputs. If not, Sim sets $\mathsf{Comp}_j = 1$. 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, during the garbling phase, for each honest virtual server V_j and each input wire w of $\mathtt{Circ}^{V_j}$, Sim doesn't follow the protocol to compute each honest member $P_{j,\alpha}$'s share of $\langle x_w \oplus \lambda_w \rangle$. Instead, Sim samples a random bit as $P_{j,\alpha}$'s share of $\langle x_w \oplus \lambda_w \rangle$ and computes his share of $\langle \lambda_w \rangle$ based on his shares of $\langle x_w \oplus \lambda_w \rangle$ and $\langle x_w \rangle$. After receiving the corrupted members' shares of $\langle x_w \oplus \lambda_w \rangle$, Sim additionally checks whether they match the inputs they committed. If not, Sim sets $\mathsf{Comp}_j = 1$. Since $P_{j,\alpha}$'s share of $\langle \lambda_w \rangle$ is sampled randomly, his share of $\langle x_w \oplus \lambda_w \rangle$ is also uniformly random. Thus, we only change the order of generating them without changing their distributions. Besides, setting $\mathsf{Comp}_j = 1$ doesn't affect the output distribution. Thus, $\mathbf{Hyb}_9$ and $\mathbf{Hyb}_8$ have the same output distribution.

$\mathbf{Hyb}_{10}$: In this hybrid, during the verification phase, for each honest virtual server V_j, Sim doesn't follow the protocol to compute each honest member $P_{j,\alpha}$'s shares of $\langle \sum_{i=1}^{k_1} s_i \cdot \boldsymbol{x}_i^{V_j} + \sum_{P_i \in \mathsf{CoreSet}} \boldsymbol{r}_1^{(i,V_j)} \rangle$, $\langle \sum_{i=1}^{k_2} s_i^{(2)} \cdot \boldsymbol{y}_i^{V_j} + \sum_{P_i \in \mathsf{CoreSet}} \boldsymbol{r}_2^{(i,V_j)} \rangle$, and $\langle \sum_{i=1}^{k_3} s_i^{(3)} \cdot \boldsymbol{z}_i^{V_j} + \sum_{i=1}^{n} \boldsymbol{r}_3^{(i,V_j)} \rangle$. Similarly for the sharings of all-zero secrets to be checked. For the last honest party $P_{i'} \in \mathcal{H} \cap \mathsf{CoreSet}$, Sim doesn't compute each honest member $P_{j,\alpha}$'s shares of $\langle \boldsymbol{r}_1^{(i',V_j)} \rangle$, $\langle \boldsymbol{r}_2^{(i',V_j)} \rangle$, and $\langle \boldsymbol{r}_3^{(i',V_j)} \rangle$. Sim computes $\sum_{i \in \mathcal{C}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{C} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle, \sum_{i \in \mathcal{C}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{C} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle, \sum_{i \in \mathcal{C}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_j^{V_j} \rangle + \sum_{i \in \mathcal{C} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle$ and $\sum_{i \in \mathcal{H}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet} \setminus \{i'\}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle$, $\sum_{i \in \mathcal{H}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{i \in \mathcal{H} \cap \mathsf{CoreSet} \setminus \{i'\}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle, \sum_{i \in \mathcal{H}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet} \setminus \{i'\}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle$ first (corrupted parties' shares are computed with the shares they receive in the sharing phase). Then, for $\sum_{i \in \mathcal{H}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle, \sum_{i \in \mathcal{H}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle, \sum_{i \in \mathcal{H}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle$, Sim randomly samples the secrets for each honest virtual server V_j based on the fact that all the virtual servers' secrets of the three sharings form a $\Sigma_{\kappa'}$-sharing, a $\Sigma_{\kappa'}^{(2)}$-sharing, and a $\Sigma_{\kappa'}^{(3)}$-sharing. Then, Sim samples the honest members $P_{j,\alpha}$'s shares of $\sum_{i \in \mathcal{H}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_\alpha} \rangle + \sum_{P_i \in \mathcal{H} \mathsf{CoreSet}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle, \sum_{i \in \mathcal{H}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle, \sum_{i \in \mathcal{H}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle$ based on the secret and corrupted member's shares (which are computed with their committed shares in $\mathsf{Sh}^{P_{j,\beta}}$ for each corrupted member $P_{j,\beta}$). For the last honest party $P_{i'}$ in $\mathsf{CoreSet}$, Sim computes the shares for each honest member $P_{j,\alpha}$ of V_j of $\langle \boldsymbol{r}_1^{(i',V_j)} \rangle$, $\langle \boldsymbol{r}_2^{(i',V_j)} \rangle$, and $\langle \boldsymbol{r}_3^{(i',V_j)} \rangle$ based on his shares of $\sum_{i \in \mathcal{H}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle, \sum_{i \in \mathcal{H}_2} s_i^{(2)} \cdot \langle \boldsymbol{y}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_2^{(i,V_j)} \rangle, \sum_{i \in \mathcal{H}_3} s_i^{(3)} \cdot \langle \boldsymbol{z}_j^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet}} \langle \boldsymbol{r}_3^{(i,V_j)} \rangle$ and $\sum_{i \in \mathcal{H}_1} s_i \cdot \langle \boldsymbol{x}_i^{V_j} \rangle + \sum_{P_i \in \mathcal{H} \cap \mathsf{CoreSet} \setminus \{i'\}} \langle \boldsymbol{r}_1^{(i,V_j)} \rangle$,

$\sum_{i\in\mathcal{H}_2} s_i^{(2)}\cdot\langle \boldsymbol{y}_i^{V_j}\rangle+\sum_{i\in\mathcal{H}\cap\mathsf{CoreSet}\setminus\{i'\}}\langle \boldsymbol{r}_2^{(i,V_j)}\rangle, \sum_{i\in\mathcal{H}_3} s_i^{(3)}\cdot\langle \boldsymbol{z}_i^{V_j}\rangle+\sum_{P_i\in\mathcal{H}\cap\mathsf{CoreSet}\setminus\{i'\}}\langle \boldsymbol{r}_3^{(i,V_j)}\rangle$. Finally, Sim computes each honest member $P_{j,\alpha}$'s share of $\langle\sum_{i=1}^{k_1} s_i\cdot\boldsymbol{x}_i^{V_j}+\sum_{P_i\in\mathsf{CoreSet}}\boldsymbol{r}_1^{(i,V_j)}\rangle$, $\langle\sum_{i=1}^{k_2} s_i^{(2)}\cdot\boldsymbol{y}_i^{V_j}+\sum_{P_i\in\mathsf{CoreSet}}\boldsymbol{r}_2^{(i,V_j)}\rangle$, and $\langle\sum_{i=1}^{k_3} s_i^{(3)}\cdot\boldsymbol{z}_i^{V_j}+\sum_{i=1}^{n}\boldsymbol{r}_3^{(i,V_j)}\rangle$ by

$$\sum_{i=1}^{k_1} s_i\cdot\langle\boldsymbol{x}_i^{V_j}\rangle+\sum_{P_i\in\mathsf{CoreSet}}\langle\boldsymbol{r}^{(i,V_j)}\rangle=\sum_{i\in\mathcal{C}_1} s_i\cdot\langle\boldsymbol{x}_i^{V_j}\rangle+\sum_{P_i\in\mathcal{C}\cap\mathsf{CoreSet}}\langle\boldsymbol{r}_1^{(i,V_j)}\rangle + \sum_{i\in\mathcal{H}_1} s_i\cdot\langle\boldsymbol{x}_i^{V_j}\rangle+\sum_{i\in\mathcal{H}\cap\mathsf{CoreSet}}\langle\boldsymbol{r}_1^{(i,V_j)}\rangle,$$

$$\sum_{i=1}^{k_2} s_i^{(2)}\cdot\langle\boldsymbol{y}_i^{V_j}\rangle+\sum_{P_i\in\mathsf{CoreSet}}\langle\boldsymbol{r}_2^{(i,V_j)}\rangle=\sum_{i\in\mathcal{C}_2} s_i^{(2)}\cdot\langle\boldsymbol{y}_i^{V_j}\rangle+\sum_{P_i\in\mathcal{C}\cap\mathsf{CoreSet}}\langle\boldsymbol{r}_2^{(i,V_j)}\rangle + \sum_{i\in\mathcal{H}_2} s_i^{(2)}\cdot\langle\boldsymbol{x}_i^{V_j}\rangle+\sum_{P_i\in\mathcal{H}\cap\mathsf{CoreSet}}\langle\boldsymbol{r}_2^{(i,V_j)}\rangle,$$

and

$$\sum_{i=1}^{k_3} s_i^{(3)}\cdot\langle\boldsymbol{z}_i^{V_j}\rangle+\sum_{P_i\in\mathsf{CoreSet}}\langle\boldsymbol{r}_3^{(i,V_j)}\rangle=\sum_{i\in\mathcal{C}_3} s_i^{(3)}\cdot\langle\boldsymbol{z}_i^{V_j}\rangle+\sum_{P_i\in\mathcal{C}\cap\mathsf{CoreSet}}\langle\boldsymbol{r}_3^{(i,V_j)}\rangle + \sum_{i\in\mathcal{H}_3} s_i^{(3)}\cdot\langle\boldsymbol{z}_i^{V_j}\rangle+\sum_{P_i\in\mathcal{H}\cap\mathsf{CoreSet}}\langle\boldsymbol{r}_3^{(i,V_j)}\rangle.$$

Since the sharings $[\boldsymbol{r}_1^{(i')}]_{\kappa'}, [\boldsymbol{r}_2^{(i')}]_{\kappa'}^{(2)}, [\boldsymbol{r}_3^{(i')}]_{\kappa'}^{(3)}$ are randomly generated based on the corrupted virtual servers' shares, we can regard that Sim emulates $P_{i'}$ to generate the secrets for each corrupted virtual server V_j first and then generate the shares for each honest virtual server V_j. Thus, the secrets of $\sum_{i\in\mathcal{H}_1} s_i\cdot\langle\boldsymbol{x}_i^{V_j}\rangle+\sum_{P_i\in\mathcal{H}\cap\mathsf{CoreSet}}\langle\boldsymbol{r}_1^{(i,V_j)}\rangle$ (resp. $\langle\sum_{i=1}^{k_2} s_i^{(2)}\cdot\boldsymbol{y}_i^{V_j}+\sum_{P_i\in\mathsf{CoreSet}}\boldsymbol{r}_2^{(i,V_j)}\rangle$ and $\langle\sum_{i=1}^{k_3} s_i^{(3)}\cdot\boldsymbol{z}_i^{V_j}+\sum_{P_i\in\mathsf{CoreSet}}\boldsymbol{r}_3^{(i,V_j)}\rangle$), which is computed by adding honest virtual server V_j's share of $[\boldsymbol{r}_1^{(i')}]_{\kappa'}$ (resp. $[\boldsymbol{r}_2^{(i')}]_{\kappa'}^{(2)}$ and $[\boldsymbol{r}_3^{(i')}]_{\kappa'}^{(3)}$) is also random when those secrets for corrupted virtual servers are fixed. Thus, we only change the order of generating each honest member $P_{j,\alpha}$'s shares of $\langle\boldsymbol{r}_1^{(i',V_j)}\rangle, \langle\boldsymbol{r}_2^{(i',V_j)}\rangle, \langle\boldsymbol{r}_3^{(i',V_j)}\rangle$ and $\sum_{i\in\mathcal{H}_1} s_i\cdot\langle\boldsymbol{x}_i^{V_j}\rangle+\sum_{P_i\in\mathcal{H}\cap\mathsf{CoreSet}}\langle\boldsymbol{r}_1^{(i,V_j)}\rangle, \sum_{i\in\mathcal{H}_2} s_i^{(2)}\cdot\langle\boldsymbol{y}_i^{V_j}\rangle+\sum_{P_i\in\mathcal{H}\cap\mathsf{CoreSet}}\langle\boldsymbol{r}_2^{(i,V_j)}\rangle, \sum_{i\in\mathcal{H}_3} s_i^{(3)}\cdot\langle\boldsymbol{z}_i^{V_j}\rangle+\sum_{P_i\in\mathcal{H}\cap\mathsf{CoreSet}}\langle\boldsymbol{r}_3^{(i,V_j)}\rangle$ for each honest virtual server V_j without changing their distributions. Similarly, for the sharings of all-zero secrets to be checked. Thus, $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ have the same output distribution.

$\mathbf{Hyb}_{11}$: In this hybrid, during the verification phase, after Sim receives $\mathsf{VS}^{P_{j,\alpha}}\oplus\mathcal{O}(\mathsf{seed}^{P'_{j,\alpha}}\|\mathsf{mid})$ from each corrupted party $P_{j,\alpha}$ of V_j, Sim additionally sets $\mathsf{Comp}_j=1$ if V_j is an honest virtual server but the corrupted servers' shares of $\sum_{i=1}^{k_1} s_i\cdot\langle\boldsymbol{x}_i^{V_j}\rangle+\sum_{P_i\in\mathsf{CoreSet}}\langle\boldsymbol{r}_1^{(i,V_j)}\rangle$, $\sum_{i=1}^{k_2} s_i^{(2)}\cdot\langle\boldsymbol{y}_i^{V_j}\rangle+\sum_{P_i\in\mathsf{CoreSet}}\langle\boldsymbol{r}_2^{(i,V_j)}\rangle$, or $\sum_{i=1}^{k_3} s_i^{(3)}\cdot\langle\boldsymbol{z}_i^{V_j}\rangle+\sum_{i=1}^{n}\langle\boldsymbol{r}_3^{(i,V_j)}\rangle$ are not correctly computed from the committed shares of $\mathsf{Sh}_i^{P_{j,\alpha}}, P_i\in\mathsf{CoreSet}$. This doesn't affect the output distribution. Similarly, for the sharings of all-zero secrets to be checked. Thus, $\mathbf{Hyb}_{11}$ and $\mathbf{Hyb}_{10}$ have the same output distribution.

$\mathbf{Hyb}_{12}$: In this hybrid, during the verification phase, after the set Ter is determined, Sim additionally sets $\mathsf{Check}_j=1$ if $V_j\in\mathsf{Ter}$ is an honest virtual party and it is checked. f there are more than $N/312$ virtual servers V_j with $\mathsf{Check}_j=1$, Sim aborts the simulation. This only affects the output distribution when there are more than $N/312$ checked honest virtual servers. Note that there are at most N virtual servers in Ter, and each of them is checked with probability $1/320$. The determination process of whether each virtual server is checked is independent. Thus, by the Chernoff bound, the probability that more than $N/312$ honest virtual parties are checked is no more than

$$e^{-\frac{(8/312)^2\cdot N/320}{2+(8/312)}}=e^{-\Omega(N)},$$

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

$\mathbf{Hyb}_{13}$: In this hybrid, during the verification phase, after the set Ter is determined, if for over $N/528$ $V_j \in \mathsf{Ver}$ it holds that $\mathsf{Comp}_j = 1$, Sim additionally sets $\mathsf{CompCheck} = 1$. If for over $N/528$ $V_j \in \mathsf{Ver}$ it holds that $\mathsf{Corr}_j = 1$, Sim additionally sets $\mathsf{CorrCheck} = 1$. If for over $N/220$ $V_j \in \mathsf{Ver}$ it holds that $\mathsf{ROT}_j = 1$, Sim additionally sets $\mathsf{OTCheck} = 1$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{13}$ and $\mathbf{Hyb}_{12}$ have the same output distribution.

$\mathbf{Hyb}_{14}$: In this hybrid, for each virtual honest virtual server V_j with $\mathsf{Check}_j = 1$, Sim doesn't follow the protocol to emulate $\mathcal{O}$ and compute each $\mathsf{Sh}_\alpha^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{IS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{MS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, and $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ for each honest party P_α and honest member $P_{j,\beta}$ of V_j. Instead, Sim first samples $\mathsf{Sh}_\alpha^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{IS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{MS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, and $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ randomly and then uses $\mathsf{Sh}_\alpha^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$ and $\mathsf{Sh}_\alpha^{P_{j,\beta}}$ to compute the corresponding output $\mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$, uses $\mathsf{IS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$ and $\mathsf{IS}^{P_{j,\beta}}$ to compute the corresponding output $\mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, uses $\mathsf{MS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$ and $\mathsf{MS}^{P_{j,\beta}}$ to compute the corresponding output $\mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, and uses $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ and $\mathsf{OS}^{P_{j,\beta}}$ to compute the corresponding output $\mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$. If some of these queries to $\mathcal{O}$ have been queried by the adversary (before the OEC process of the seeds), Sim aborts the simulation. Note that if these queries have not been queried, the outputs $\mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$, $\mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, and $\mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ are uniformly random, so are $\mathsf{Sh}_\alpha^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{IS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{MS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, and $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$. Under this condition, we only change the order of generating $\mathsf{Sh}_\alpha^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{IS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{MS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ and $\mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$, $\mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, $\mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ without changing their distributions. Thus, the output distribution only changes when some of the queries have been queried by the adversary. However, the only messages sent to $\mathcal{A}$ that are computed with these seeds are $\mathsf{Sh}_\alpha^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{IS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{MS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, and $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$, which are uniformly random when the seeds are not queried. Thus, for each query from $\mathcal{A}$, the probability that this query matches the seeds and the message IDs is at most $1/2^\kappa$. Take the union bound of all $\mathsf{poly}(\kappa)$ possible queries from the adversary, the probability that $\mathcal{A}$ has queried these values is negligible. Thus, the distributions of $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{13}$ are computationally indistinguishable.

Note that the seeds $\mathsf{seed}_\alpha^{P_{j,\beta}}$, $\mathsf{seed}^{P_{j,\beta}}$, and $\mathsf{seed}_0^{P_{j,\beta}}$ for honest $P_\alpha, P_{j,\beta}$ are not used until the set of checked virtual parties is determined, we delay the generation of them until Sim gets the set of checked virtual parties in future hybrids.

$\mathbf{Hyb}_{15}$: In this hybrid, for each share $\boldsymbol{v}_{i,\alpha}^{V_j}$ generated by an honest party P_i for an honest party $P_{j,\alpha}$, Sim doesn't follow the protocol to compute $\boldsymbol{v}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ and emulate $\mathcal{F}_{\mathsf{AVID}}$ to send it to $\mathcal{A}$. Instead, Sim samples $\boldsymbol{v}_{i,\alpha}^{V_j} \oplus \mathcal{O}(k_{i,j_\alpha} \| \mathsf{mid})$ randomly. In addition, Sim doesn't follow the protocol to compute each $\mathsf{Sh}_\alpha^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_\alpha^{P_{j,\beta}} \| \mathsf{mid})$, $\mathsf{IS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$, and $\mathsf{MS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}^{P_{j,\beta}} \| \mathsf{mid})$ for honest parties $P_\alpha, P_{j,\beta}$ such that the honest virtual server V_j is not checked. Sim also doesn't follow the protocol to compute $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ for each pair of honest parties $P_\alpha, P_{j,\beta}$ such that the honest virtual server V_j is not in Ter. Sim also samples random strings as them. The same as in $\mathbf{Hyb}_{14}$, these pair-wise keys and seeds are only used as a part of the queries to the random oracle, and the distribution only changes when $\mathcal{A}$ has queried them. For the same reason as in $\mathbf{Hyb}_{14}$, the probability that the output distribution changes is negligible. Thus, the distributions of $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$ are computationally indistinguishable.

Note that the pair-wise keys k_{i,j_α} for honest $P_i, P_{j_\alpha} = P_{j,\alpha}$, the seeds $\mathsf{seed}_\alpha^{P_{j,\beta}}, \mathsf{seed}^{P_{j,\beta}}$ for honest $P_\alpha, P_{j,\beta}$ such that the honest virtual server V_j is not checked, and the seed $\mathsf{seed}_0^{P_{j,\beta}}$ for each honest party $P_{j,\beta}$ such that the honest virtual server V_j is not in Ter are not used in the whole simulation process, Sim doesn't generate them in future hybrids.

$\mathbf{Hyb}_{16}$: In this hybrid, during the verification phase, after doing the checks, if $\mathsf{OTCheck} = 1$, Sim aborts the simulation. This only changes the distribution when some corrupted members of V_j do not correctly commit their outputs from $\mathcal{F}_{\mathsf{ROT}}$ for over $N/220$ honest virtual parties $V_j \in \mathsf{Ter}$ after simulating the sharing phase but the verification passes. The probable ways that there is a checked $V_j \in \mathsf{Ter}$ of $\mathsf{ROT}_j = 1$ that passes the check are

1. For the execution of $\mathcal{F}_{\mathsf{ROT}}$ between $(P_{j,\alpha}, P_{j,\beta})$ where $P_{j,\alpha}$ is honest and $P_{j,\beta}$ is corrupted, $P_{j,\beta}$ sends $b^{(\alpha,\beta)}, r^{(\alpha,\beta)}_{b^{(\alpha,\beta)}}$ to $\mathcal{F}_{\mathsf{ROT}}$ but he correctly commits $(1 \oplus b^{(\alpha,\beta)}, r^{(\alpha,\beta)}_{1\oplus b^{(\alpha,\beta)}})$ as his output from $\mathcal{F}_{\mathsf{ROT}}$.
2. For the execution of $\mathcal{F}_{\mathsf{ROT}}$ between $(P_{j,\beta}, P_{j,\alpha})$ where $P_{j,\alpha}$ is honest and $P_{j,\beta}$ is corrupted, $P_{j,\beta}$ sends $r^{(\alpha,\beta)}_0, r^{(\alpha,\beta)}_1$ to $\mathcal{F}_{\mathsf{ROT}}$ but he correctly guesses $b^{(\alpha,\beta)}$ and commits $r^{(\alpha,\beta)}_{b^{(\alpha,\beta)}}$ and another value which is not equal to $r^{(\alpha,\beta)}_{1\oplus b^{(\alpha,\beta)}}$ as his output from $\mathcal{F}_{\mathsf{ROT}}$.

For the first way, the adversary should commit $r^{(\alpha,\beta)}_{1\oplus b^{(\alpha,\beta)}}$, which is computed from a randomly sampled $(\kappa-1)$-bit string $r^{(\alpha,\beta)}_1 \oplus r^{(\alpha,\beta)}_0 \oplus x^{(\alpha)}$, the value $r^{(\alpha,\beta)}_{b^{(\alpha,\beta)}}$, and the share $x^{(\alpha)}$ of $\langle \boldsymbol{x} \rangle$ for the corresponding execution of Π_{Multi}. Note that $r^{(\alpha,\beta)}_1 \oplus r^{(\alpha,\beta)}_0 \oplus x^{(\alpha)}$ is sampled randomly after $\mathsf{IS}^{P_{j,\beta}}$ is determined. Thus, the probability that the adversary correctly commits $r^{(\alpha,\beta)}_{1\oplus b^{(\alpha,\beta)}}$ is $2^{-\Omega(\kappa)}$, which is negligible.

For the second way, the adversary should guess the random bit $b^{(\alpha,\beta)}$ correctly. $b^{(\alpha,\beta)}$ is computed from a random sampled bit $y^{\alpha} \oplus b^{(\alpha,\beta)}$ and the share $y^{(\alpha)}$ of $\langle \boldsymbol{y} \rangle$ for the corresponding execution of Π_{Multi}. Note that $y^{\alpha} \oplus b^{(\alpha,\beta)}$ is sampled randomly after $\mathsf{IS}^{P_{j,\beta}}$ is determined. Thus, the probability that the adversary correctly commits $b^{(\alpha,\beta)}$ is $1/2$. Thus, for each virtual server V_j with $\mathsf{ROT}_j = 1$, if $\mathsf{Check}_j = 1$, the honest party will figure out the malicious behavior of the adversary with a probability of at least $1/2$. The distribution only changes when the set of over $N/220$ indices j with $V_j \in \mathsf{Ter}$ and $\mathsf{ROT}_j = 1$ satisfies that each virtual server V_j with j in this set is not checked. The probability is

$$\left(\frac{1}{320} \cdot \frac{1}{2}\right)^{\frac{N}{220}} = e^{-\Omega(N)}.$$

Recall that $N = \Theta(n + \kappa)$, the probability is negligible.

Thus, the distributions of $\mathbf{Hyb}_{16}$ and $\mathbf{Hyb}_{15}$ are statistically close.

$\mathbf{Hyb}_{17}$: In this hybrid, during the verification phase, after doing the checks, if $\mathsf{CompCheck} = 1$, Sim aborts the simulation. This only changes the distribution when $\mathsf{Comp}_j = 1$ for over $N/528$ honest virtual servers $V_j \in \mathsf{Ter}$ but the verification passes with $\mathsf{CompCheck} = 0$. Note that $\mathsf{Comp}_j = 1$ only happens when some messages sent by corrupted members of V_j do not match the committed inputs or shares. Thus, if V_j is checked, the check won't pass, so the distribution only changes when the set over $N/528$ indices j with $V_j \in \mathsf{Ter}$ and $\mathsf{Comp}_j = 1$ satisfies that each virtual server V_j with j in this set is not checked. The probability is

$$\left(\frac{1}{320}\right)^{\frac{N}{528}} = e^{-\Omega(N)}.$$

Recall that $N = \Theta(n + \kappa)$, the probability is negligible. Thus, the distributions of $\mathbf{Hyb}_{17}$ and $\mathbf{Hyb}_{16}$ are statistically close.

$\mathbf{Hyb}_{18}$: In this hybrid, during the verification phase, after doing the checks, if $\mathsf{CorrCheck} = 1$, Sim aborts the simulation. This only changes the distribution when $\mathsf{Corr}_j = 1$ for over $N/528$ honest virtual servers $V_j \in \mathsf{Ter}$ but the verification passes with $\mathsf{CorrCheck} = 0$. Note that the only possibility to cause $\mathsf{Corr}_j = 1$ after simulating the sharing phase is that the committed inputs of corrupted members of V_j do not match the generated shares by honest parties or the committed inputs of corrupted members of V_j do not match the committed shares generated by corrupted parties. For all these reasons, if V_j is checked, the honest parties will abort the protocol. Thus, the output distribution only changes when the set of $N/528$ indices j with $V_j \in \mathsf{Ter}$ and $\mathsf{Corr}_j = 1$ satisfies that each virtual server V_j with j in this set is not checked. For the same reason as in $\mathbf{Hyb}_{17}$, the distributions of $\mathbf{Hyb}_{18}$ and $\mathbf{Hyb}_{17}$ are statistically close.

$\mathbf{Hyb}_{19}$: In this hybrid, during the verification phase, after doing the checks, Sim chooses a subset $\mathcal{H}_{\mathsf{vir}} \subset \mathsf{Ter}$ of $|\mathsf{Ter} \cup \mathsf{Corr}| - N/20$ honest virtual servers, where each $V_j \in \mathcal{H}_{\mathsf{vir}}$ satisfies $\mathsf{Corr}_j = \mathsf{Comp}_j = \mathsf{ROT}_j = \mathsf{Check}_j = 0$. Then, Sim checks whether the shares of the virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of are valid shares of each Σ-sharing, $\Sigma^{(2)}$-sharing, and each $\Sigma^{(3)}$-sharing that is verified in the first step of the verification phase and is generated by a corrupted party. If not, Sim aborts the simulation. Otherwise, Sim chooses a valid sharing as the sharing shared by the corrupted party and reconstructs the secret.

Assume that the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of a Σ-sharing are not valid. Assume that the random coefficient on this sharing in $[\tau]_{\kappa'} = \sum_{i=1}^{k_1} s_i \cdot [\boldsymbol{x}_i]_{\kappa'} + \sum_{P_i \in \mathsf{CoreSet}} [\boldsymbol{r}_1^{(i)}]_{\kappa'}$ is $s \in \mathbb{F}_{2^{\kappa'}}$. If $s_1, \ldots, s_{k_1} \in \mathbb{F}_{2^{\kappa'}}$ are all truly random, we can sample s after the invalid sharing is fixed. If there exists $s_0 \neq s_0' \in \mathbb{F}_{2^{\kappa'}}$ such that $s = s_0$ and $s = s_0'$ both lead to a valid $[\tau]_{\kappa'}$, then the invalid sharing (which has been embedded in a $\Sigma_{\times\kappa'}$-sharing) is $(s_0 - s_0')^{-1}$ times a valid $\Sigma_{\times\kappa'}$-sharing, which must be a valid $\Sigma_{\times\kappa'}$-sharing, and this leads to a contradiction. Thus, there is only one element $s_0 \in \mathbb{F}_{2^{\kappa'}}$ that can make $[\tau]_{\kappa'}$ pass the check. The probability is $2^{-\kappa'}$. Considering the union bound for no more than 2^N possible choices of $\mathcal{H}_{\mathsf{vir}}$, the probability is still no more than $2^{N-\kappa'} = 2^{-\kappa}$ if $s_1, \ldots, s_{k_1} \in \mathbb{F}_{2^{\kappa'}}$ are all truly random, which is negligible. Since $s_1, \ldots, s_{k_1}$ are not truly random only when s has been queried by the adversary to the random oracle before it is sampled by Sim. Since s is sampled randomly from $\mathbb{F}_{2^\kappa}$, and Sim can only send $\mathsf{poly}(\kappa)$ queries, the probability that s is among them is negligible.

Similarly, when the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of a $\Sigma^{(2)}$-sharing, a $\Sigma^{(3)}$-sharing, or a $\Sigma/\Sigma^{(2)}/\Sigma^{(3)}$-sharing with an all-zero secret are not valid, the verification only passes with a negligible probability. Therefore, the distribution only changes with a negligible probability. Thus, the distributions of $\mathbf{Hyb}_{19}$ and $\mathbf{Hyb}_{18}$ are computationally indistinguishable.

$\mathbf{Hyb}_{20}$: In this hybrid, for each virtual honest virtual server $V_j \in \mathsf{ter}$ with $\mathsf{Check}_j = 0$, Sim doesn't follow the protocol to emulate $\mathcal{O}$ and compute $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ for each honest party P_α and honest member $P_{j,\beta}$ of V_j. Instead, Sim first samples $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ randomly and then uses $\mathsf{OS}^{P_{j,\beta}} \oplus \mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$ and $\mathsf{OS}^{P_{j,\beta}}$ to compute the corresponding output $\mathcal{O}(\mathsf{seed}_0^{P_{j,\beta}} \| \mathsf{mid})$. If some of these queries to $\mathcal{O}$ have been queried by the adversary (before the OEC process of the seeds), Sim aborts the simulation. For the same reason as in $\mathbf{Hyb}_{14}$, the distributions of $\mathbf{Hyb}_{20}$ and $\mathbf{Hyb}_{19}$ are computationally indistinguishable.

$\mathbf{Hyb}_{21}$: In this hybrid, for each random $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharing (including the interleaved secret sharings of them) generated by an honest party, Sim does not generate them at the beginning of the sharing phase. Instead, for those honest virtual servers $V_j \in \mathsf{Ter}$ with $\mathsf{Check}_j = 1$, Sim generates the shares for them after the set of checked virtual servers is determined. For other honest virtual servers in $\mathsf{Ter} \backslash \mathcal{H}_{\mathsf{vir}}$, Sim generates V_j's shares of these sharings based on the checked honest virtual servers' shares and the corrupted virtual servers' shares. For other honest virtual servers, Sim generates their shares based on other virtual servers' shares after the verification phase (at the beginning of the evaluation phase). Finally, Sim samples the shares for honest members of the additive sharings of these virtual servers' shares based on corrupted members' shares and the secrets. Besides, Sim samples the local randomness of each honest member $P_{j,\alpha}$ in emulating each $V_j \in \mathcal{H}_{\mathsf{vir}}$ and each honest virtual server out of Ter after the verification phase instead of in the sharing phase, and the computation of $P_{j,\alpha}$ is delayed to be performed in the evaluation phase instead of in the garbling phase. Note that for all these sharings, the set of corrupted servers' shares and the shares of honest virtual servers of $\mathsf{Ter} \backslash \mathcal{H}_{\mathsf{vir}}$ are independent of the secret, first sampling the shares for them and then sampling the shares for other virtual servers won't affect the output distribution. Besides, the shares and randomness for each honest member $P_{j,\alpha}$ of V_j in $\mathcal{H}_{\mathsf{vir}}$ or out of Ter are not used in the simulation before the evaluation phase. Therefore, delaying the generation won't affect the output distribution. Thus, $\mathbf{Hyb}_{21}$ and $\mathbf{Hyb}_{20}$ have the same output distribution.

Note that for each honest virtual server $V_j \notin \mathsf{Ter}$, the shares for each honest member $P_{j,\alpha}$ of the sharings generated by each honest party P_i are not used in the simulation. We don't generate them in future hybrids.

$\mathbf{Hyb}_{22}$: In this hybrid, for each wire w in each honest virtual server $V_j \in \mathcal{H}_{\mathsf{Vir}}$'s local circuit $\mathtt{Circ}^{V_j}$, Sim additionally follows the execution of Π_0 to compute the value v_w of w (where each reconstruction of $\Sigma^{(2)}$-sharing is done from the shares of virtual servers in $\mathcal{H}_{\mathsf{Vir}}$. From the property of $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharings introduced in Section 3.2, the secrets are determined by these shares). 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, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$:

1. For $i = 1, \ldots, \mathsf{rec}$:

   - If the receiver of $[\boldsymbol{s}_i]$ is an honest client, Sim doesn't generate the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each the $\Sigma^{(3)}$-sharings generated in the sharing phase that is used to prepare $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)}$ ($\alpha =$

$1, \ldots, \kappa, \beta = 1, 2, 3$) first and then computes each share for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ by himself. Instead, Sim randomly samples $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$ first and then samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ based on the shares for virtual servers of $\mathcal{C}_{\mathsf{Vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$. Then, Sim computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}, [\boldsymbol{s}_i]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ for each $\beta = 1, 2, 3$.

- If the receiver of $[\boldsymbol{s}_i]^{(2)}$ is a virtual server in $\mathcal{H}_{\mathsf{vir}}$, Sim doesn't generate the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each the $\Sigma^{(3)}$-sharings generated in the sharing phase that is used to prepare $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ ($\alpha = 1, \ldots, \alpha, \beta = 1, 2, 3$) first and then computes each share for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ for each honest member $P_{j,\beta}$ of V_j by himself. Instead, Sim randomly samples $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$ first and then samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ based on the shares for virtual servers of $\mathcal{C}_{\mathsf{Vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$. Then, Sim computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}, [\boldsymbol{s}_i]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ for each honest member $P_{j,\beta}$ of V_j.
- If the receiver of $[\boldsymbol{s}_i]^{(2)}$ is a virtual server of $\mathsf{Ter} \backslash \mathcal{H}_{\mathsf{Vir}}$, Sim just follows the protocol to receive honest parties' shares from corrupted parties.

Then Sim follows the computation process of each $Y^{P_{j,\beta}}_{(i-1)\ell^2+a, s^{V_j}_{i,a}}$ in the protocol to compute each $Y^{P_{j,\beta}}_{(i-1)\ell^2+a, s^{V_j}_{i,a}}$ with each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ generated by the honest clients and the first servers emulating virtual servers in $\mathcal{H}_{\mathsf{vir}}$ being regarded as $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$. For the other label $Y^{P_{j,\beta}}_{(i-1)\ell^2+a, 1 \oplus s^{V_j}_{i,a}}$, Sim still follows the protocol to compute it.

2. For each honest member $P_{j,\alpha}$ of V_j each wire w in each $\mathtt{Circ}^{V_j}$ that is not an input wire of the circuit and is not an output wire of an XOR gate or an output gate, Sim samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string as $k^{P_{j,\alpha}}_{w, v_w \oplus \lambda_w}$. For each input wire of $\mathtt{Circ}^{V_j}$, Sim samples a random $(\kappa-1)$-bit string as $k^{P_{j,\alpha}}_{w, v_w \oplus \lambda_w}$. Then Sim computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$ for all these wires. After all the ciphertexts are generated, Sim samples a random $(\kappa-1)$-bit string as $\Delta^{P_{j,\alpha}}$. Then, for each wire w in $\mathtt{Circ}^{V_j}$ that is not an output wire of an output gate, Sim computes $k^{P_{j,\alpha}}_{w, 1 \oplus v_w \oplus \lambda_w} = k^{P_{j,\alpha}}_{w, v_w \oplus \lambda_w} \oplus \Delta^{P_{j,\alpha}}$.

3. Sim maintains a set Q_1. For each AND gate g in $\mathtt{Circ}^{V_j}$ with input wire a, b, when Sim emulates an honest member $P_{j,\alpha}$ of V_j to compute his shares of the additive sharings of the ciphertexts of each gate except those computed with $\{k^{P_{j,i}}_{a, v_a \oplus \lambda_a}, k^{P_{j,i}}_{b, v_b \oplus \lambda_b}\}_{i=1}^{3}$, Sim checks whether the query to the random oracle $\mathcal{O}$ has been queried before. If true, Sim aborts the simulation. Otherwise, Sim adds the query to Q_1.

4. Sim maintains a set Q_2. For each input wire w of an output gate in $\mathtt{Circ}^{V_j}$, and for all $i_2 \in \{0, 1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, when an honest member $P_{j,\alpha}$ of V_j computes his shares of the additive sharings of $\mathsf{ct}^{P_{j,\beta}}_{w, i_2}$ for each $\beta = 1, 2, 3$, Sim checks whether the query to the random oracle $\mathcal{O}$ has been queried before. If true, Sim aborts the simulation. Otherwise, Sim adds the query to Q_2.

5. For each AND gate g in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, Sim doesn't follow the protocol to compute each honest member $P_{j,\alpha}$'s shares of the additive sharings of the ciphertexts except $A^{P_{j,\beta}}_{g, 2(v_a \oplus \lambda_a) + (v_b \oplus \lambda_b)}$ for $\beta = 1, 2, 3$. Instead, Sim samples 3 random κ-bit strings as the ciphertexts and samples honest members' shares of the additive sharings of all the ciphertexts based on the secrets and corrupted members' shares. Sim computes the output of $\mathcal{O}$ to the queries that are used to generate these ciphertexts based on the random strings and the wire labels of wire o.

6. For each output gate in $\mathtt{Circ}^{V_j}$ indexed $1, \ldots, \ell^2\mathsf{rec}$ with input wire w, Sim doesn't follow the protocol to compute each honest member $P_{j,\alpha}$'s shares of the additive sharings of the ciphertexts $\mathsf{ct}_{w,1\oplus v_w\oplus\lambda_w}^{P_{j,\beta}}$ for $\beta = 1, 2, 3$. Instead, Sim samples a random $c\ell\kappa$-bit string as the ciphertexts and randomly samples honest members' shares of the additive sharings of all the ciphertexts based on the secrets and corrupted members' shares. Sim computes the output of $\mathcal{O}$ to the queries that are used to generate these ciphertexts based on the random strings and the output labels.
7. For each input wire w of an output gate in $\mathtt{Circ}^{V_j}$, Sim doesn't follow the protocol to compute each honest member $P_{j,\alpha}$'s share of $\langle\lambda_w\rangle$. Instead, Sim randomly samples honest members' shares based on λ_w and the corrupted members' shares of $\langle\lambda_w\rangle$.
8. If V_j is the receiver of $[\boldsymbol{s}_i]^{(2)}$ in Π_0 and the β-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_η in $\mathtt{Circ}^{V_j}$, Sim doesn't follow the protocol to compute $\mathsf{ct}_{j_\eta,1\oplus s_{\beta,\alpha}}^{(\beta,\alpha)}$ for each honest member $P_{j,\alpha}$ and corrupted member $P_{j,\beta}$ of V_j. Instead, Sim samples a random κ-bit string as $\mathsf{ct}_{j_\eta,1\oplus s_{\beta,\alpha}}^{(\beta,\alpha)}$.

To prove that the distributions of $\mathbf{Hyb}_{23}$ and $\mathbf{Hyb}_{22}$ are computationally indistinguishable, we additionally construct the following hybrids between $\mathbf{Hyb}_{22}$ and $\mathbf{Hyb}_{23}$.

$\mathbf{Hyb}_{23.0}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim computes each honest member $P_{j,\alpha}$'s shares of the garbled circuit of $\mathtt{Circ}^{V_j}$ by computing the garbled sub-circuits $\mathtt{Circ}_1^{V_j}, \ldots, \mathtt{Circ}_{\mathsf{rec}}^{V_j}$ in order. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{23.0}$ and $\mathbf{Hyb}_{22}$ have the same distribution.

$\mathbf{Hyb}_{23.1.1}$: In this hybrid, Sim additionally computes the share for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{s}_1]^{(2)}$ by using the input labels associated with the input of V_j to evaluate the garbled gates of $\mathtt{Circ}_1^{V_j}$ following Steps 2.(b) and 2.(c) of the evaluation phase. Here the input of the local computation (performed via the inner protocol) for V_j to the receiver is shared by an additive sharing among $P_{j,1}, P_{j,2}, P_{j,3}$, and all the secrets of these sharings have been computed by Sim. We use these secrets as the input and preprocessing data to compute the input labels. We denote the result of the computation be $\overline{[\boldsymbol{s}_1]^{(2)}}$. Note that the input of $\mathtt{Circ}_1^{V_j}$ completely comes from the preprocessing and input phases of Π_0, and the computation process of S_j's share of $[\boldsymbol{s}_1]^{(2)}$ in Π_0 is identical to $\mathtt{Circ}_1^{V_j}$. To show that $[\boldsymbol{s}_1]^{(2)} = \overline{[\boldsymbol{s}_1]^{(2)}}$, we only need to show that the secrets of the additive sharings of the inputs of V_j computed by Sim are the same as the outputs to V_j from the preprocessing and input phases in Π_0. Since $V_j \in \mathcal{H}_{\mathsf{vir}}$, the corrupted parties just follow the protocol to distribute the sharings for the parties emulating V_j. From the correctness of the multiparty garbling process of Π_{in}, the result $\overline{[\boldsymbol{s}_1]^{(2)}}$ is the same as $[\boldsymbol{s}_1]^{(2)}$ from the execution of Π_0.

In addition, if the receiver of $[\boldsymbol{s}_1]^{(2)}$ is an honest client, Sim doesn't generate the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each the $\Sigma^{(3)}$-sharings generated in the sharing phase that is used to prepare each $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)}$ first and then computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}]^{(3)}$ by himself. Instead, Sim generates $[\boldsymbol{s}_1]^{(2)}$ first and then samples the whole sharing $[\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}]^{(3)}$ based on the shares for virtual servers of $\mathcal{C}_{\mathsf{vir}}$ of $[\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}]^{(3)} = [\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)} + [\boldsymbol{s}_1]^{(2)} \otimes [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$ and the secret $\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}$ for each $\beta = 1, 2, 3$. Then, Sim computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)}$ based on their shares of $[\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}]^{(3)}, [\boldsymbol{s}_1]^{(2)}, [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$. Similarly, if the receiver of $[\boldsymbol{s}_1]^{(2)}$ is a virtual server in $\mathcal{H}_{\mathsf{vir}}$, Sim doesn't generate the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each the $\Sigma^{(3)}$-sharings generated in the sharing phase that is used to prepare $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)}$ for each honest member $P_{j,\beta}$ first and then computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}]^{(3)}$ by himself. Instead, Sim generates $[\boldsymbol{s}_1]^{(2)}$ first and then samples the whole sharing $[\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}]^{(3)}$ based on the shares for virtual servers of $\mathcal{C}_{\mathsf{vir}}$ of $[\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}]^{(3)} = [\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)} + [\boldsymbol{s}_1]^{(2)} \otimes [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$ and the secret $\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}$ for each honest member $P_{j,\beta}$ of V_j. Then, Sim computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)}$ based on their shares of $[\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}]^{(3)}, [\boldsymbol{s}_1]^{(2)}, [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$ for each honest member $P_{j,\beta}$ of V_j. Then for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim follows the computational process to compute each $Y_{a,s_{1,a}}^{P_{j,\beta},V_j}$ in the protocol to compute each $Y_{a,s_{1,a}}^{P_{j,\beta},V_j}$ with each $[\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}]^{(3)}$ whose secrets are held by the honest clients and the honest members emulating

virtual servers in $\mathcal{H}_{\mathsf{vir}}$ being regarded as $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)} + [\boldsymbol{s}_1]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$. For the other label $Y^{P_{j,\beta}}_{a,1\oplus s^{V_j}_{1,a}}$, Sim still follows the protocol to compute it.

Since the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ generated by the honest clients and the honest members emulating virtual servers in $\mathcal{H}_{\mathsf{vir}}$ are sampled randomly based on the shares for virtual servers out of $\mathcal{H}_{\mathsf{vir}}$ and the secrets, and the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta}]^{(3)}$ can be computed only based on their shares of shares of $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}], [\boldsymbol{s}_1]^{(2)}$ and the secret, the only restrictions on the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}]^{(3)}$ are the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ and the distribution of $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}$. Therefore, we only change the order of generating the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}]^{(3)}$ and $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}$ whose secrets are held by an honest party without changing their distributions. Thus, $\mathbf{Hyb}_{23.1.1}$ and $\mathbf{Hyb}_{23.0}$ have the same distribution.

$\mathbf{Hyb}_{23.1.2}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim doesn't follow the protocol to compute each honest member $P_{j,\alpha}$'s shares of the garbled circuit for $\mathtt{Circ}_1^{V_j}$. Instead, for each wire w in $\mathtt{Circ}_1^{V_j}$ that is not an input wire of the circuit and is not an output wire of an XOR gate or an output gate, Sim samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string as $k^{P_{j,\alpha}}_{w,v_w\oplus\lambda_w}$. For each input wire of $\mathtt{Circ}^{V_j}$, Sim samples a random $(\kappa-1)$-bit string as $k^{P_{j,\alpha}}_{w,v_w\oplus\lambda_w}$. Then Sim computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$ for these wires. For each wire w in $\mathtt{Circ}_1^{V_j}$ that is not an output wire of an output gate, Sim computes $k^{P_{j,\alpha}}_{w,1\oplus v_w\oplus\lambda_w} = k^{P_{j,\alpha}}_{w,v_w\oplus\lambda_w} \oplus \Delta^{P_{j,\alpha}}$. Since λ_w is a uniformly sampled bit, $v_w \oplus \lambda_w$ is also a uniformly random bit. Therefore, we only change the order of generating $v_w \oplus \lambda_w$ and λ_w without changing their distributions. Similarly, if $v_w \oplus \lambda_w = 1$, we only change the order of generating $k_{w,0}$ and $k_{w,1}$ without changing their distributions. If $v_w \oplus \lambda_w = 0$, we doesn't change anything on $k_{w,0}$ and $k_{w,1}$. Thus, $\mathbf{Hyb}_{23.1.2}$ and $\mathbf{Hyb}_{23.1.1}$ have the same output distribution.

$\mathbf{Hyb}_{23.1.3}$: In this hybrid, Sim maintains a set Q_1. For each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, for each AND gate g in $\mathtt{Circ}_1^{V_j}$ with input wire a, b, and for the ciphertexts of this gate except those computed with $\{k^{P_{j,i}}_{a,v_a\oplus\lambda_a}, k^{P_{j,i}}_{b,v_b\oplus\lambda_b}\}_{i=1}^3$, Sim checks whether each query of each honest server $P_{j,\alpha}$ to the random oracle $\mathcal{O}$ has been queried before. If true, Sim aborts the simulation. Otherwise, Sim adds the query to Q_1. Note that all the queries to the random oracle by the honest servers are distinct, and the adversary's queries to the random oracle before the encryption are fixed when the wire labels and the value $\Delta^{P_{j,\alpha}}$ are generated. Since each query made to the random oracle by $P_{j,\alpha}$ contains either of the $(\kappa-1)$-bit strings $k^{P_{j,\alpha}}_{a,1\oplus v_a\oplus\lambda_a}$ or $k^{P_{j,\alpha}}_{b,1\oplus v_b\oplus\lambda_b}$ with $k^{P_{j,\alpha}}_{a,1\oplus v_a\oplus\lambda_a} - k^{P_{j,\alpha}}_{a,v_a\oplus\lambda_a} = k^{P_{j,\alpha}}_{b,1\oplus v_b\oplus\lambda_b} - k^{P_{j,\alpha}}_{b,v_b\oplus\lambda_b} = \Delta^{P_{j,\alpha}}$ which is uniformly random, the probability that each query made by the adversary is one of the queries made by the honest server is negligible. Taking the union bound of all the $\mathsf{poly}(\kappa)$ queries made by the adversary, the probability that some query has been queried (either by the honest server or by the adversary) is negligible. Thus, the distributions of $\mathbf{Hyb}_{23.1.3}$ and $\mathbf{Hyb}_{23.1.2}$ are computationally indistinguishable.

$\mathbf{Hyb}_{23.1.4}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, for each AND gate g in $\mathtt{Circ}_1^{V_j}$ with input wire a, b and output wire o, and for the ciphertexts of this gate except those computed with $\{k^{P_{j,i}}_{a,v_a\oplus\lambda_a}, k^{P_{j,i}}_{b,v_b\oplus\lambda_b}\}_{i=1}^3$, Sim samples random κ-bit strings as the ciphertexts and samples honest members' shares of the additive sharings of the ciphertexts based on the secret and corrupted members' shares. While emulating $\mathcal{O}$, for each $i_0, i_1 \in \{0,1\}$ such that $(i_0, i_1) \neq (v_a\oplus\lambda_a, v_b\oplus\lambda_b)$, Sim_2 computes the output of $\mathcal{O}(k^{P_{j,\alpha}}_{a,i_0} \| i_0 \| k^{P_{j,\alpha}}_{b,i_1} \| i_1 \| 1 \| j \| i \| g)$ for each honest member $P_{j,\alpha}$ of V_j and $i = 1, 2, 3$ based on the random strings and the wire labels of wire o. Note that the only difference between $\mathbf{Hyb}_{23.1.4}$ and $\mathbf{Hyb}_{23.1.3}$ is the way we decide the output for queries in Q_1. Since the ciphertext is randomly sampled, the XOR of the ciphertext and the message m is also uniformly random for any κ-bit string m. In particular, when Sim does not abort the simulation, queries in Q_1 have not been queried before. Thus, $\mathbf{Hyb}_{23.1.4}$ and $\mathbf{Hyb}_{23.1.3}$ have the same output distribution.

$\mathbf{Hyb}_{23.1.5}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim changes the order of sampling random κ-bit strings as the ciphertexts of this gate except those computed with $\{k^{P_{j,i}}_{a,v_a\oplus\lambda_a}, k^{P_{j,i}}_{b,v_b\oplus\lambda_b}\}_{i=1}^3$ and sampling $\Delta^{P_{j,\alpha}}$ for honest members $P_{j,\alpha}$ of V_j to decide the queries to $\mathcal{O}$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{23.1.5}$ and $\mathbf{Hyb}_{23.1.4}$ have the same

output distribution.

$\mathbf{Hyb}_{20.1.6}$: In this hybrid, Sim maintains a set Q_2. For each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, for each input wire w of an output gate in $\mathtt{Circ}_1^{V_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, while computing each honest member $P_{j,\alpha}$'s share of $\langle \mathsf{ct}_{w,i_2}^{P_{j,\beta}} \rangle$ for each $\beta = 1,2,3$, Sim checks whether the query made by $P_{j,\alpha}$ to the random oracle $\mathcal{O}$ has been queried before. If true, Sim aborts the simulation. Otherwise, Sim adds the query to Q_2. Note that each query made to the random oracle contains a $(\kappa-1)$-bit string $k_{w,1\oplus v_w\oplus\lambda_w}^{P_{j,\alpha}}$ with $k_{w,1\oplus v_w\oplus\lambda_w}^{P_{j,\alpha}} - k_{w,v_w\oplus\lambda_w}^{P_{j,\alpha}} = \Delta^{P_{j,\alpha}}$, for the same reason in $\mathbf{Hyb}_{23.1.3}$, the probability that some query has been queried by the adversary is negligible. Thus, the distributions of $\mathbf{Hyb}_{23.1.6}$ and $\mathbf{Hyb}_{23.1.5}$ are computationally indistinguishable.

$\mathbf{Hyb}_{23.1.7}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, for each input wire w of an output gate in $\mathtt{Circ}_1^{V_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, Sim samples a random $c\ell\kappa$-bit string as the ciphertext $\mathsf{ct}_{w,i_2}^{P_{j,\beta}}$ and computes honest members' shares of the additive sharing of the ciphertext based on the secret and corrupted members' shares. While emulating $\mathcal{O}$, Sim computes the output based on the random strings and the output labels. Note that the only difference between $\mathbf{Hyb}_{23.1.7}$ and $\mathbf{Hyb}_{23.1.6}$ is the way we decide the output for queries in Q_2. Since $\mathsf{ct}_{w,i_2}^{P_{j,\beta}}$ is randomly sampled, $\mathsf{ct}_{w,i_2}^{P_{j,\beta}} \oplus m$ is also uniformly random for any $c\ell\kappa$-bit string m. In particular, when Sim does not abort the simulation, queries in Q_2 have not been queried before. Thus, $\mathbf{Hyb}_{23.1.7}$ and $\mathbf{Hyb}_{23.1.6}$ have the same output distribution.

$\mathbf{Hyb}_{23.1.8}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, for each input wire w of an output gate in $\mathtt{Circ}_1^{V_j}$, Sim doesn't follow the protocol to compute honest members' shares of $\langle \lambda_w \rangle$. Instead, Sim samples them based on λ_w and the corrupted members' shares of $\langle \lambda_w \rangle$. Since honest members' shares of $\langle \lambda_w \rangle$ are computed by his shares of those wires that are not an output wire of an XOR gate or an output gate, where honest members' shares of $\langle \lambda_w \rangle$ are generated based on λ_w and the corrupted members' shares of $\langle \lambda_w \rangle$. Therefore, for each input wire w of an output gate, we just change the order of generating for each input wire w of an output gate and honest members' shares of $\langle \lambda_w \rangle$. Thus, $\mathbf{Hyb}_{23.1.8}$ and $\mathbf{Hyb}_{23.1.7}$ have the same output distribution.

$\mathbf{Hyb}_{23.1.9}$: In this hybrid, if the receiver of $[\boldsymbol{s}_1]^{(2)}$ is a virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$ and the β-th bit of $\boldsymbol{s}_1$ is used as an input wire with index j_β in $\mathtt{Circ}^{V_j}$, when each honest member $P_{j,\alpha}$ of V_j computes $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,\alpha)}$, Sim checks whether the query to the random oracle $\mathcal{O}$ has been queried before. If true, Sim aborts the simulation. Otherwise, Sim adds the query to Q_1. Since $\boldsymbol{r}_{1\oplus s_{1,\beta},\beta,\alpha}$ is generated randomly in $\{0,1\}^\kappa$, the probability that some query has been queried (either by the honest parties or by the adversary) is negligible. Thus, the distributions of $\mathbf{Hyb}_{23.1.9}$ and $\mathbf{Hyb}_{23.1.8}$ are computationally indistinguishable.

$\mathbf{Hyb}_{23.1.10}$: In this hybrid, if the receiver of $[\boldsymbol{s}_1]^{(2)}$ is a virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$ and the β-th bit of $\boldsymbol{s}_1$ is used as an input wire with index j_β in $\mathtt{Circ}^{V_j}$, Sim doesn't follow the protocol to compute $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,\alpha)}$ for each honest member $P_{j,\alpha}$ of V_j. Instead, Sim samples a random κ-bit string as $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,\alpha)}$. While emulating $\mathcal{O}$, Sim computes the output of $\mathcal{O}(\boldsymbol{r}_{1\oplus s_{1,\beta},\beta} \| s_{1,\beta} \| 1 \| \alpha \| \beta \| j_\beta)$ based on $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,\alpha)}$ and $k_{w_{j_\beta},1\oplus v_{w_{j_\beta}}^{P_{j,\alpha}}\oplus\lambda_{w_{j_\beta}}} \| (1 \oplus v_{w_{j_\beta}} \oplus \lambda_{w_{j_\beta}})$. Note that the only difference between $\mathbf{Hyb}_{23.1.10}$ and $\mathbf{Hyb}_{23.1.9}$ is the way we decide the output for queries in Q_1. Since $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,\alpha)}$ is randomly sampled, $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,\alpha)} \oplus m$ is also uniformly random for any κ-bit string m. In particular, when Sim does not abort the simulation, queries in Q_1 have not been queried before. Thus, $\mathbf{Hyb}_{23.1.10}$ and $\mathbf{Hyb}_{23.1.9}$ have the same output distribution.

$\mathbf{Hyb}_{23.1.11}$: In this hybrid, for virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim changes the order of sampling a random κ-bit string as the ciphertext $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,\alpha)}$ for each honest member $P_{j,\alpha}$ of V_j and sampling $\boldsymbol{r}_{1\oplus s_{1,\beta},\beta,\alpha}$ to decide the queries to $\mathcal{O}$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{23.1.11}$ and $\mathbf{Hyb}_{23.1.10}$ have the same output distribution.

$\mathbf{Hyb}_{23.1.12}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim changes the order of sampling random $c\ell\kappa$-bit strings as the ciphertexts $\mathsf{ct}_{w,i_2}^{P_{j,\beta}}$ for each $\beta = 1,2,3$ and each input wire w of an output gate in $\mathtt{Circ}_1^{V_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$ and sampling $\Delta^{P_{j,\alpha}}$ for each honest member $P_{j,\alpha}$ of V_j to decide the queries to $\mathcal{O}$. Since these two steps are both local computations, this doesn't affect the output

distribution. Thus, $\mathbf{Hyb}_{23.1.12}$ and $\mathbf{Hyb}_{23.1.11}$ have the same output distribution.

Note that for each member $P_{j,\alpha}$ of each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, $\Delta^{P_{j,\alpha}}$ is not used before all the ciphertexts of the gates in $\mathtt{Circ}_1^{V_j}$ are generated. Sim delays the generating of $\Delta^{P_{j,\alpha}}$ after the garbling of $\mathtt{Circ}_1^{V_j}$ is completed.

Similarly, for each $\gamma = 2, \ldots, \mathsf{rec}$ we can define $\mathbf{Hyb}_{23.\gamma.1}, \ldots, \mathbf{Hyb}_{23.\gamma.12}$.

$\mathbf{Hyb}_{23.\gamma.1}$: In this hybrid, Sim additionally computes the share for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{s}_\gamma]^{(2)}$ by using the input labels associated with the input of V_j to evaluate the garbled gates of $\mathtt{Circ}_\gamma^{V_j}$ following Steps 2.(b) and 2.(c) of the evaluation phase. Here the input of V_j is shared by an additive sharing among $P_{j,1}, P_{j,2}, P_{j,3}$, and all the secrets of these sharings have been computed by Sim. We use these secrets as the input and preprocessing data to compute the input labels. We denote the result of the computation be $\overline{[\boldsymbol{s}_\gamma]^{(2)}}$. Note that the input of $\mathtt{Circ}_\gamma^{V_j}$ completely comes from the preprocessing and input phase of Π_0 and the reconstructions of $\overline{[\boldsymbol{s}_1]^{(2)}}, \ldots, \overline{[\boldsymbol{s}_{\gamma-1}]^{(2)}}$, and since $\overline{[\boldsymbol{s}_i]^{(2)}} = [\boldsymbol{s}_i]^{(2)}$ for each $i = 1, \ldots, \gamma-1$ and the computation process of S_j's share of $[\boldsymbol{s}_\gamma]^{(2)}$ is the same in Π_0 with $\mathtt{Circ}_\gamma^{V_j}$, for the same reason in $\mathbf{Hyb}_{23.1.1}$, the result $\overline{[\boldsymbol{s}_\gamma]^{(2)}}$ is the same as $[\boldsymbol{s}_\gamma]^{(2)}$ from the execution of Π_0.

In addition, if the receiver of $[\boldsymbol{s}_\gamma]^{(2)}$ is an honest client, Sim doesn't generate the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each the $\Sigma^{(3)}$-sharings generated in the sharing phase that is used to prepare each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ first and then computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}]^{(3)}$ by himself. Instead, Sim generates $[\boldsymbol{s}_\gamma]^{(2)}$ first and then samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}]^{(3)}$ based on the shares for virtual servers of $\mathcal{C}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}_\gamma]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}$ for each $\beta = 1, 2, 3$. Then, Sim computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}]^{(3)}, [\boldsymbol{s}_\gamma]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$. Similarly, if the receiver of $[\boldsymbol{s}_\gamma]^{(2)}$ is a virtual server in $\mathcal{H}_{\mathsf{vir}}$, Sim doesn't generate the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each the $\Sigma^{(3)}$-sharings generated in the sharing phase that is used to prepare each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ first and then computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}]^{(3)}$ for each honest member $P_{j,\beta}$ by himself. Instead, Sim generates $[\boldsymbol{s}_\gamma]^{(2)}$ first and then samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,1}]^{(3)}$ based on the shares for virtual servers of $\mathcal{C}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,1}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},1}]^{(3)} + [\boldsymbol{s}_\gamma]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},1} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},1}]$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}$. Then, Sim computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}]^{(3)}, [\boldsymbol{s}_\gamma]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ for each honest member $P_{j,\alpha}$ of V_j. Then for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim follows the computational process to compute each $Y^{P_{j,\beta}}_{a,s^{V_j}_{\gamma,a}}$ in the protocol to compute each $Y^{P_{j,\beta}}_{a,s^{V_j}_{\gamma,a}}$ with each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}]^{(3)}$ generated by the honest clients and the honest members of virtual servers in $\mathcal{H}_{\mathsf{vir}}$ regarded as $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}_\gamma]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$. For the other label $Y^{P_{j,\beta}}_{a,1\oplus s^{V_j}_{\gamma,a}}$, Sim still follows the protocol to compute it. For the same reason in $\mathbf{Hyb}_{23.1.1}$, $\mathbf{Hyb}_{23.\gamma.1}$ and $\mathbf{Hyb}_{23.(\gamma-1).12}$ have the same distribution.

$\mathbf{Hyb}_{20.\gamma.2}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim doesn't follow the protocol to compute the honest members' shares of the garbled circuit for $\mathtt{Circ}_\gamma^{V_j}$. Instead, for each wire w in $\mathtt{Circ}_\gamma^{V_j}$ that is not an input wire of the circuit and is not an output wire of an XOR gate or an output gate, Sim samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string as $k^{P_{j,\alpha}}_{w,v_w\oplus\lambda_w}$ for each honest member $P_{j,\alpha}$ of V_j. Then Sim computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$ for these wires. For each wire w in $\mathtt{Circ}_\gamma^{V_j}$ that is not an output wire of an output gate, Sim computes $k^{P_{j,\alpha}}_{w,1\oplus v_w\oplus\lambda_w} = k^{P_{j,\alpha}}_{w,v_w\oplus\lambda_w} \oplus \Delta^{P_{j,\alpha}}$ for each honest member $P_{j,\alpha}$ of V_j. For the same reason in $\mathbf{Hyb}_{23.1.2}$, $\mathbf{Hyb}_{23.\gamma.2}$ and $\mathbf{Hyb}_{23.\gamma.1}$ have the same output distribution.

$\mathbf{Hyb}_{20.\gamma.3}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, for each AND gate g in $\mathtt{Circ}_\gamma^{V_j}$ with input wire a, b, and for the ciphertexts of this gate except those computed with $\{k^{P_{j,i}}_{a,v_a\oplus\lambda_a}, k^{P_{j,i}}_{b,v_b\oplus\lambda_b}\}_{i=1}^3$, Sim checks whether each query of each honest member $P_{j,\alpha}$ of V_j to the random oracle $\mathcal{O}$ has been queried before. If true, Sim aborts the simulation. Otherwise, Sim adds the query to Q_1. For the same reason in $\mathbf{Hyb}_{23.1.3}$, the distributions of $\mathbf{Hyb}_{23.\gamma.3}$ and $\mathbf{Hyb}_{23.\gamma.2}$ are computationally indistinguishable.

$\mathbf{Hyb}_{20.\gamma.4}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, for each AND gate g in $\mathtt{Circ}_\gamma^{V_j}$ with input wire

a, b and output wire o, and for the ciphertexts of this gate except those computed with $\{k^{P_{j,i}}_{a,v_a\oplus\lambda_a}, k^{P_{j,i}}_{b,v_b\oplus\lambda_b}\}^3_{i=1}$, Sim samples random κ-bit strings as the ciphertexts and computes honest members' shares of the additive sharings of the ciphertexts based on the secret and corrupted members' shares. While emulating $\mathcal{O}$, for each $i_0, i_1 \in \{0,1\}$ such that $(i_0, i_1) \neq (v_a \oplus \lambda_a, v_b \oplus \lambda_b)$, Sim computes the output of $\mathcal{O}(k^{P_{j,\alpha}}_{a,i_0} \| i_0 \| k^{P_{j,\alpha}}_{b,i_1} \| i_1 \| 1 \| j \| i \| g)$ for each honest member $P_{j,\alpha}$ of V_j and $i = 1, 2, 3$ based on the random strings and the wire labels of wire o. For the same reason in $\mathbf{Hyb}_{23.1.4}$, $\mathbf{Hyb}_{23.\gamma.4}$ and $\mathbf{Hyb}_{23.\gamma.3}$ have the same output distribution.

$\mathbf{Hyb}_{23.\gamma.5}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim changes the order of sampling random κ-bit strings as the ciphertexts of this gate except those computed with $\{k^{P_{j,i}}_{a,v_a\oplus\lambda_a}, k^{P_{j,i}}_{b,v_b\oplus\lambda_b}\}^3_{i=1}$ and sampling $\Delta^{P_{j,\alpha}}$ for each honest member $P_{j,\alpha}$ of V_j to decide the queries to $\mathcal{O}$. For the same reason in $\mathbf{Hyb}_{23.1.5}$, $\mathbf{Hyb}_{23.\gamma.5}$ and $\mathbf{Hyb}_{23.\gamma.4}$ have the same output distribution.

$\mathbf{Hyb}_{23.\gamma.6}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, for each input wire w of an output gate in $\mathtt{Circ}^{V_j}_\gamma$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, while computing honest members' shares of $\langle \mathsf{ct}^{P_{j,\beta}}_{w,i_2} \rangle$ for each $\beta = 1, 2, 3$, Sim checks whether the query made by honest members of V_j to the random oracle $\mathcal{O}$ has been queried before. If true, Sim aborts the simulation. Otherwise, Sim adds the query to Q_2. For the same reason in $\mathbf{Hyb}_{23.1.6}$, the distributions of $\mathbf{Hyb}_{23.\gamma.6}$ and $\mathbf{Hyb}_{23.\gamma.5}$ are computationally indistinguishable.

$\mathbf{Hyb}_{23.\gamma.7}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, for each input wire w of an output gate in $\mathtt{Circ}^{V_j}_\gamma$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, Sim samples a random $c\ell\kappa$-bit string as the ciphertext $\mathsf{ct}^{P_{j,\beta}}_{w,i_2}, \beta = 1, 2, 3$ and computes honest members' shares of the additive sharing of the ciphertext based on the secret and corrupted members' shares. While emulating $\mathcal{O}$, Sim computes the output based on the random strings and the output labels. For the same reason in $\mathbf{Hyb}_{23.1.7}$, $\mathbf{Hyb}_{23.\gamma.7}$ and $\mathbf{Hyb}_{23.\gamma.6}$ have the same output distribution.

$\mathbf{Hyb}_{23.\gamma.8}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, for each input wire w of an output gate in $\mathtt{Circ}^{V_j}_\gamma$, Sim doesn't follow the protocol to compute honest members' shares of $\langle \lambda_w \rangle$. Instead, Sim samples them based on λ_w and the corrupted members' shares of $\langle \lambda_w \rangle$. For the same reason in $\mathbf{Hyb}_{23.1.8}$, $\mathbf{Hyb}_{23.\gamma.8}$ and $\mathbf{Hyb}_{23.\gamma.7}$ have the same output distribution.

$\mathbf{Hyb}_{23.\gamma.9}$: In this hybrid, if the receiver of $[\boldsymbol{s}_\gamma]^{(2)}$ is a virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$ and the β-th bit of $\boldsymbol{s}_\gamma$ is used as an input wire with index j_β in $\mathtt{Circ}^{V_j}$, when each honest party $P_{j,\alpha}$ computes $\mathsf{ct}^{(\gamma,\alpha)}_{j_\beta, 1\oplus s_{\gamma,\beta}}$, Sim checks whether the query to the random oracle $\mathcal{O}$ has been queried before. If true, Sim aborts the simulation. Otherwise, Sim adds the query to Q_1. For the same reason in $\mathbf{Hyb}_{23.1.9}$, the distributions of $\mathbf{Hyb}_{23.\gamma.9}$ and $\mathbf{Hyb}_{23.\gamma.8}$ are computationally indistinguishable.

$\mathbf{Hyb}_{23.\gamma.10}$: In this hybrid, if the receiver of $[\boldsymbol{s}_\gamma]^{(2)}$ is a virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$ and the β-th bit of $\boldsymbol{s}_\gamma$ is used as an input wire with index j_β in $\mathtt{Circ}^{V_j}$, Sim doesn't follow the protocol to compute $\mathsf{ct}^{(\gamma,\alpha)}_{j_\beta, 1\oplus s_{\gamma,\beta}}$. Instead, Sim samples a random κ-bit string as $\mathsf{ct}^{(\gamma,\alpha)}_{j_\beta, 1\oplus s_{\gamma,\beta}}$. While emulating $\mathcal{O}$, Sim computes the output of $\mathcal{O}(\boldsymbol{r}_{1\oplus s_{\gamma,\beta},\beta} \| s_{\gamma,\beta} \| \gamma \| \alpha \| \beta \| j_\beta)$ based on $\mathsf{ct}^{(\gamma,\alpha)}_{j_\beta, 1\oplus s_{\gamma,\beta}}$ and $k_{w_{j_\beta}, 1\oplus v^{P_{j,\alpha}}_{w_{j_\beta}} \oplus \lambda_{w_{j_\beta}}} \| (1 \oplus v_{w_{j_\beta}} \oplus \lambda_{w_{j_\beta}})$. For the same reason in $\mathbf{Hyb}_{23.1.10}$, $\mathbf{Hyb}_{23.\gamma.10}$ and $\mathbf{Hyb}_{23.\gamma.9}$ have the same output distribution.

$\mathbf{Hyb}_{23.\gamma.11}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim changes the order of sampling a random κ-bit string as the ciphertext $\mathsf{ct}^{(\gamma,\alpha)}_{j_\beta, 1\oplus s_{\gamma,\beta}}$ and sampling $\boldsymbol{r}_{1\oplus s_{\gamma,\beta},\beta,\alpha}$ for each honest member $P_{j,\alpha}$ of V_j to decide the queries to $\mathcal{O}$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{23.\gamma.11}$ and $\mathbf{Hyb}_{23.\gamma.10}$ have the same output distribution.

$\mathbf{Hyb}_{23.\gamma.12}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim changes the order of sampling random $c\ell\kappa$-bit strings as the ciphertexts $\mathsf{ct}^{P_{j,\beta}}_{w,i_2}$ for each $\beta = 1, 2, 3$ and each input wire w of an output gate in $\mathtt{Circ}^{V_j}_\gamma$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$ and sampling $\Delta^{P_{j,\alpha}}$ for each honest member $P_{j,\alpha}$ of to decide the queries to $\mathcal{O}$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{23.\gamma.12}$ and $\mathbf{Hyb}_{23.\gamma.11}$ have the same output distribution.

Note that $\mathbf{Hyb}_{23.\mathsf{rec}.12}$ is just $\mathbf{Hyb}_{23}$, we conclude that the distributions of $\mathbf{Hyb}_{23}$ and $\mathbf{Hyb}_{22}$ are computationally indistinguishable.

Note that for each honest member $P_{j,\alpha}$ of each virtual server $V_j \in \mathcal{H}_{\mathsf{Vir}}$, $\Delta^{P_{j,\alpha}}$ is not used before all the

ciphertexts of the gates in $\mathtt{Circ}_\gamma^{V_j}$ are generated. Sim delays the generating of $\Delta^{P_{j,\alpha}}$ after the garbling of $\mathtt{Circ}_\gamma^{V_j}$ is completed.

Also note that if the receiver of $[\boldsymbol{s}_i]^{(2)}$ is an honest client, $\boldsymbol{r}^{(\alpha)}_{\mathbf{1}\oplus\boldsymbol{s}_i,\beta}$ for $\alpha = 1,\ldots,\kappa$ and $\beta = 1,2,3$ are not used until the whole garbled circuit of V_j is generated. Sim generates them after the whole garbled circuit is generated in future hybrids to decide the set Q_1.

$\mathbf{Hyb}_{24}$: In this hybrid, for each AND gate g in each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$'s local circuit $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, Sim doesn't compute the output of $\mathcal{O}$ to the queries that are used to generate these ciphertexts based on the random strings and the wire labels of wire o. Similarly, for each output gate in $\mathtt{Circ}^{V_j}$ with input wire w, Sim_2 doesn't compute the output of $\mathcal{O}$ to the queries that are used to generate these ciphertexts based on the random strings and the output labels. Instead, Sim honestly emulates the random oracles. In particular, Sim no longer checks whether the queries to the random oracles to compute the ciphertexts for V_j have been queried before. We prove that the distributions of $\mathbf{Hyb}_{24}$ and $\mathbf{Hyb}_{23}$ are computationally indistinguishable.

For the sake of contradiction, assume that there exists an adversary $\mathcal{A}_1$ such that $\mathbf{Hyb}_{24}$ and $\mathbf{Hyb}_{23}$ are computationally distinguishable. Let Q_1, Q_2 be the set of queries to the random oracle respectively when Sim_2 is computing the ciphertexts for V_j that are randomly generated in the last hybrid. Now we argue that, with non-negligible probability, at least one query in Q_1 or Q_2 has been queried. Suppose this is not the case. Note that all queries in Q_1 are distinct. Then, by assumption, with overwhelming probability, no query in Q_1 has been queried, and all queries in Q_1 are distinct. Similar for Q_2. In this case, the only difference between $\mathbf{Hyb}_{23}$ and $\mathbf{Hyb}_{24}$ is that we do not explicitly compute the output to each query in Q_1, Q_2. Since no query in Q_1, Q_2 has been queried, this makes no difference in the output distribution. Then it shows that $\mathbf{Hyb}_{24}$ and $\mathbf{Hyb}_{23}$ are computationally indistinguishable, which leads to a contradiction.

Thus, with non-negligible probability, at least one query in Q_1 or Q_2 has been queried in $\mathbf{Hyb}_{24}$. However, each query in Q_1, Q_2 either contains $k_{w,1\oplus v_w\oplus\lambda_w}$ for a wire w in $\mathtt{Circ}^{V_j}$ for some $V_j \in \mathcal{H}_{\mathsf{vir}}$ or contains $\boldsymbol{r}_{1\oplus s_{i,\eta},\eta,1}$ for some honest virtual server receiver $R_i \in \mathcal{H}_{\mathsf{vir}}$. Suppose a query contains $k^{P_{j,\alpha}}_{w,1\oplus v_w\oplus\lambda_w}$ for a wire w in an honest server V_j's circuit $\mathtt{Circ}^{V_j}$ for an honest member $P_{j,\alpha}$ of V_j. Since $\Delta^{P_{j,\alpha}}$ is generated after the garbled circuit is generated, and it is not used to compute any transcript sent to $\mathcal{A}$, the queries are independent of $\Delta^{P_{j,\alpha}}$. Therefore, $k^{P_{j,\alpha}}_{w,1\oplus v_w\oplus\lambda_w} = k^{P_{j,\alpha}}_{w,v_w\oplus\lambda_w} \oplus \Delta^{P_{j,\alpha}}$ only has $2^{-\kappa+1}\cdot\mathsf{poly}(\kappa)$ probability to be queried by $\mathcal{A}$, which is negligible. Similarly, if a query contains $\boldsymbol{r}_{1\oplus s_{i,\eta},\eta,1}$ for some honest virtual server receiver $V_j \in \mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{s}_i]^{(2)}$, since $\boldsymbol{r}_{1\oplus s_{i,\eta},\eta,\alpha}$ is not used to compute any transcript sent to $\mathcal{A}$ before all the ciphertexts are generated, it can be generated after the garbled circuit is generated, and thus the probability that it is queried by $\mathcal{A}$ is also negligible. Thus, the distributions of $\mathbf{Hyb}_{24}$ and $\mathbf{Hyb}_{23}$ are computationally indistinguishable.

Note that if the receiver of $[\boldsymbol{s}_i]^{(2)}$ is an honest client, we only need each $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$ and we don't need $\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}, \boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta}$ for $\beta = 1,2,3$ for the simulation, and we also don't need honest members' shares of $\{[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]\}_{\alpha=1}^{\kappa}$ for $\beta = 1,2,3$ in the simulation. Sim doesn't generate them in future hybrids.

$\mathbf{Hyb}_{25}$: In this hybrid, for each virtual server V_j that is not in $\mathcal{C}_{\mathsf{Vir}}$ or $\mathcal{H}_{\mathsf{Vir}}$ and $i = 1,\ldots,\mathsf{rec}$, if V_j is the receiver of $[\boldsymbol{s}_i]^{(2)}$ in Π_0 and the β-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_η in $\mathtt{Circ}^{V_j}$, Sim doesn't follow the protocol to compute $\mathsf{ct}^{(\beta,\alpha)}_{j_\eta,1\oplus s_{\beta,\alpha}}$ for each honest member $P_{j,\alpha}$ and corrupted member $P_{j,\beta}$ of V_j. Instead, Sim samples a random κ-bit string as $\mathsf{ct}^{(\beta,\alpha)}_{j_\eta,1\oplus s_{\beta,\alpha}}$. Since the output of the inner protocol of V_j is not used in the simulation, the above changes do not affect the output distribution. Thus, $\mathbf{Hyb}_{25}$ and $\mathbf{Hyb}_{24}$ have the same distribution.

$\mathbf{Hyb}_{26}$: In this hybrid, since all the transcripts between honest and corrupted parties generated by Sim can be generated from the transcripts between honest parties (honest clients and honest servers in $\mathcal{H}_{\mathsf{Vir}}$) and corrupted parties obtained in the execution of Π_0 (only using input sharings of parties in $\mathsf{CoreSet}$), where the input and preprocessing data have been prepared in a secret-shared way among the members of each virtual server. Sim runs the honest servers corresponding to virtual servers in $\mathcal{H}_{\mathsf{Vir}}$ first to obtain all the transcripts, and then Sim uses them to generate the output. In addition, honest parties don't follow the protocol Π to compute their output. Instead, they follow Π_0 to get their output. Since the value $\boldsymbol{s}_i$ sent and the preprocessing and input data of Π_0 and Π are the same, the computation of honest parties' outputs in

the two protocols is completely the same. Therefore, we only change the way of generating the output of Sim without changing their distributions. Thus, $\mathbf{Hyb}_{26}$ and $\mathbf{Hyb}_{25}$ have the same distribution.

$\mathbf{Hyb}_{27}$: In this hybrid, Sim doesn't run the honest servers (corresponding to virtual servers in $\mathcal{H}_{\mathsf{Vir}}$) in Π_0 to get the transcripts between honest and corrupted parties in Π_0 and use them to compute all the transcripts between honest and corrupted parties in Π. Instead, Sim samples the shares of $[\boldsymbol{s}_i]^{(2)}$ for honest virtual servers in $\mathcal{H}_{\mathsf{Vir}}$ randomly based on the shares for virtual servers in $\mathcal{C}_{\mathsf{Vir}}$. Note that all the reconstructions in Π_0 come from executions of $\Pi_{\mathsf{Transpose}}$ (each $\Sigma^{(2)}$-sharing to be reconstructed in Π_0 is a sharing $[\boldsymbol{y}_i]^{(2)}$ of an execution of $\Pi_{\mathsf{Transpose}}$), Sim uses them to compute the corresponding sharing $[\boldsymbol{r}_i]^{(2)}$ based on the equation $[\boldsymbol{y}_i]^{(2)} = [F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{N\ell}) + \boldsymbol{r}_i]^{(2)}$ in $\Pi_{\mathsf{Transpose}}$. Note that for each virtual server V_j, the sharing $[\boldsymbol{r}_i]^{(2)}$ is prepared by preparing random sharings $[\boldsymbol{r}_{i,1}]^{(2)}, [\boldsymbol{r}_{i,2}]^{(2)}, [\boldsymbol{r}_{i,3}]^{(2)}$ and add them together. We only need to show that the sharing $[\boldsymbol{r}_{i,\alpha}]^{(2)}$ for honest member $P_{j,\alpha}$ is random based on the shares for virtual servers in $\mathcal{C}_{Vir}$, and then so is $[\boldsymbol{r}_i]^{(2)}$. Note that Sim samples the sharing of $[\boldsymbol{v}'_i]^{(2)}_a, [\boldsymbol{o}'_i]^{(2)}$ based on corrupted servers' shares for each execution of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$. Let $\mathcal{C}'$ be the set of indices of corrupted parties in $\mathsf{CoreSet}$ and $\mathcal{H}'$ be the set of indices of honest parties in $\mathsf{CoreSet}$. Then let $\mathcal{N}_{\mathcal{C}'}$ denote the sub-matrix of $\mathcal{N}$ containing columns with indices in $\mathcal{C}'$, and $\mathcal{N}_{\mathcal{H}'}$ denote the sub-matrix containing columns with indices in $\mathcal{H}'$. We have

$$\begin{pmatrix} [\boldsymbol{v}_1]^{(2)}_a \\ [\boldsymbol{v}_2]^{(2)}_a \\ \vdots \\ [\boldsymbol{v}_{t+1}]^{(2)}_a \end{pmatrix} = \mathcal{N} \cdot \begin{pmatrix} [\boldsymbol{v}'_1]^{(2)}_a + [\boldsymbol{o}_1]^{(2)}_a \\ [\boldsymbol{v}'_2]^{(2)}_a + [\boldsymbol{o}_2]^{(2)}_a \\ \vdots \\ [\boldsymbol{v}'_{2t+1}]^{(2)}_a + [\boldsymbol{o}_{2t+1}]^{(2)}_a \end{pmatrix} = \mathcal{N}_{\mathcal{C}'} \cdot \left([\boldsymbol{v}_j]^{(2)}_a\right)_{j\in\mathcal{C}'} + \mathcal{N}_{\mathcal{H}'} \cdot \left([\boldsymbol{v}'_j]^{(2)}_a + [\boldsymbol{o}_j]^{(2)}_a\right)_{j\in\mathcal{H}'}.$$

Recall that $\mathcal{N}^T$ is a Vandermonde matrix of size $(2t+1)\times(t+1)$. Therefore $\mathcal{N}^T_{\mathcal{H}'}$ is a Vandermonde matrix of size $(t+1)\times(t+1)$, which is invertible. Thus, there is a bijective map from $\left([\boldsymbol{v}_j]^{(2)}_a\right)_{j\in\mathcal{H}'}$ and $\left([\boldsymbol{v}'_i]^{(2)}_a + [\boldsymbol{o}'_i]^{(2)}_a\right)_{i=1}^{t+1}$. Recall that $\left([\boldsymbol{o}_j]^{(2)}_a\right)_{j\in\mathcal{H}'}$ are sampled randomly based on corrupted servers' shares and the secrets, so $\left([\boldsymbol{v}'_i]^{(2)}_a + [\boldsymbol{o}'_i]^{(2)}_a\right)_{j\in\mathcal{H}'}$ are also completely random when corrupted servers' shares are fixed, and so are $\left([\boldsymbol{v}'_i]^{(2)}_a + [\boldsymbol{o}'_i]^{(2)}_a\right)_{i=1}^{t+1}$. Therefore, the sharing $[\boldsymbol{y}_i]^{(2)}$ for each execution of $\Pi_{\mathsf{Transpose}}$ is also completely random based on corrupted servers' shares. Thus, we only change the order of generating the shares for virtual servers in $\mathcal{H}_{\mathsf{Vir}}$ of the sharing $[\boldsymbol{y}_i]^{(2)}$ and $[\boldsymbol{r}_i]^{(2)}$ for each execution of $\Pi_{\mathsf{Transpose}}$ without changing their distributions. Thus, $\mathbf{Hyb}_{27}$ and $\mathbf{Hyb}_{26}$ have the same output distribution.

$\mathbf{Hyb}_{27}$: In this hybrid, honest parties get their outputs from $\mathcal{F}$ instead of following Π_0 to compute them, where the input of corrupted parties comes from the secret of each input sharing generated by corrupted parties in $\mathsf{CoreSet}$, where the secrets are reconstructed from the shares of virtual servers in $\mathcal{H}_{\mathsf{Vir}}$. The correctness of Π_0 can be easily verified, which implies that the outputs from the two hybrids are the same. Thus, $\mathbf{Hyb}_{27}$ and $\mathbf{Hyb}_{26}$ have the same distribution.

Note that $\mathbf{Hyb}_{27}$ is the ideal-world scenario, Π computes $\mathcal{F}$ with computational security. □

J Cost Analysis of the Main Protocol

J.1 Analysis of Communication Complexity

Setup Phase. Recall that the circuit size of computing $\mathcal{F}_{\mathsf{Setup}}$ is independent of C. Therefore, the communication cost of the setup phase is $O(\mathsf{poly}(n,\kappa))$ bits.

Sharing Phase. We analyze the communication cost of the sharing phase step by step as follows:

1. **Preparing Random $\Sigma^{(2)}$-Sharings.** The subprotocol $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ is executed $2G_AN(\ell+3)/a(t+1)k^2 + 3W_O/a(t+1)k$ times in parallel in this step. All the shares sent from a party P_i to a party P_j can be sent together via a single invocation of $\mathcal{F}_{\mathsf{AVID}}$. Thus, $\mathcal{F}_{\mathsf{AVID}}$ is called $3Nn$ times, the message length of each invocation of $\mathcal{F}_{\mathsf{AVID}}$ is $2\ell^2 a \cdot 2G_AN(\ell+3)/a(t+1)k^2 + 3W_O/a(t+1)k = O(|C|/Nn)$.

Thus, the cost of these executions of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ is $O(|C| + \mathsf{poly}(n, \kappa))$. In addition, each sharing $[\boldsymbol{v}'_\alpha]^{(2)}$ shared by P_α in each execution of $\Pi^{(2)}_{\mathsf{RandShare\text{-}Share}}$ to prepare sharings of form $[\boldsymbol{r}_{j,i}]^{(2)}$, the secret should be reconstructed to $P_{j,i}$. This requires another $3Nn$ invocation of $\mathcal{F}_{\mathsf{AVID}}$, each of message length $2\ell^2 a \cdot 2G_A/a(t+1)k^2 = O(|C|/Nn)$, resulting in extra communication cost of $O(|C| + \mathsf{poly}(n, \kappa))$ bits. Similarly for the sharings prepared for output wires. Thus, the total communication cost of this step is $O(|C| + \mathsf{poly}(n, \kappa))$ bits.

2. **Preprocessing for the Verification of Sharings.** In this step, the servers distributes $2n$ $\Sigma_{\times\kappa'}$, $\Sigma^{(2)}_{\times\kappa'}$, $\Sigma^{(3)}_{\times\kappa'}$-sharings of size $O(N\kappa')$ via AVID. $\mathcal{F}_{\mathsf{AVID}}$ is invoked $3nN$ times, each of message length $O(\kappa')$. Thus, the communication cost of this step is $O(nN\kappa') = O(\mathsf{poly}(n, \kappa))$ bits.

3. **Sharing Inputs.** For each group of $k = O(N)$ input wires of C, a party needs to distribute a Σ-sharing of $O(N)$ bits via AVID in this step. The communication cost of this step is $O(|C| + \mathsf{poly}(n, \kappa))$ bits.

4. **Preparing for the Garbling of Local Circuits.** In this step, $\Pi^{(1)}_{\mathsf{RandShare\text{-}Share}}$ and $\Pi^{(3)}_{\mathsf{RandShare\text{-}Share}}$ are executed $3\kappa\mathsf{rec}/a(t+1)$ times in parallel respectively, where the secret of each prepared sharing needs to be reconstructed to a party. Similar to Step 1, the total cost is $O(\mathsf{rec}N + \mathsf{poly}(n, \kappa))$ bits. Note that $\mathsf{rec} = O(|C|/N)$ in Π_0, the total communication cost is $O(|C| + \mathsf{poly}(n, \kappa))$ bits.

5. **Committing Sharings.** In this step, the set of sharings $\mathsf{Sh}_i^{P_{j,\alpha}}$ needs to be committed via AVID for each $i = 1, \ldots, n$, $j = 1, \ldots, N$, and $\alpha = 1, 2, 3$. $\mathcal{F}_{\mathsf{AVID}}$ needs to be invoked $3Nn$ times. The total data that needs to be committed is at most the total cost of previous steps, i.e. $O(|C| + \mathsf{poly}(n, \kappa))$. Thus, the total cost of this step is still $O(|C| + \mathsf{poly}(n, \kappa))$ bits.

6. **Determining the Core Set.** This step only contains an execution of Π^{Q}_{ACS}, which requires communication of $O(\mathsf{poly}(n, \kappa))$ bits.

7. **Committing Local Inputs.** The data that needs to be committed are all the received sharings (plus some of their secrets) in previous steps, the local randomness generated by each party, and the outputs of base ROT instances. The received sharings are of size $O(\mathsf{poly}(n, \kappa))$. The local randomness contains a wire label for each input wire of each virtual server's local circuit and each output wire of each AND gate of the local circuit. Note that the total number of AND operations required in Π_0 is $O(|C|)$, thus the total size of the labels is $O(|C|\kappa)$. Other terms in the local randomness are of size $O(\mathsf{poly}(n, \kappa))$. $\mathcal{F}_{\mathsf{AVID}}$ needs to be invoked $3N$ times. Thus, the total communication cost of this step is $O(|C|\kappa + \mathsf{poly}(n, \kappa))$ bits.

As we analyzed above, the total communication cost of the sharing phase is $O(|C|\kappa + \mathsf{poly}(n, \kappa))$ bits.

Local Computation Phase. This phase only contains local computation of the parties and requires no communication.

Garbing Phase. We analyze the communication cost of the garbling phase step by step as follows. We analyze the cost of Π_{in}:

1. **OT Extension.** From the result from [KOS15], the OT extension from $\kappa - 1$ instances of ROTs (of message length $\kappa - 1$) to $13 \cdot G_A^{V_j} + 6\ell W_O^{V_j}$ instances of ROTs requires communication of $O((13 \cdot G_A^{V_j} + 6\ell W_O^{V_j}) \cdot (\kappa - 1)) = O(G_A^{V_j} + W_O^{V_j})$. Recall that $\sum_{j=1}^{N} G_A^{V_j} = O(|C|)$ and $W_O^{V_j} = \mathsf{rec} \cdot \ell^2 = O(|C|/N)$, the total communication of this step is $O(|C|)$ bits for the executions of all the virtual servers' inner protocols.

2. **Handling XOR gates.** This step only contains local computation of the parties and requires no communication.

3. **Computing 1-Labels.** This step only contains local computation of the parties and requires no communication.

4. **Handling AND Gates.** In this step, communication only happens during the executions of Π_{Mult}. The communication cost of each execution of Π_{Mult} is $O(\kappa)$, and the servers need to run Π_{Mult} $O(G_A^{V_j})$ times for each virtual server V_j, resulting in a total communication of $O(|C|\kappa)$ bits.

5. **Handling Output Gates.** In this step, communication only happens during the executions of Π_{Mult}. The communication cost of each execution of Π_{Mult} is $O(\kappa)$, and the servers need to run Π_{Mult} $O(W_O^{V_j})$ times for each virtual server V_j, resulting in a total communication of $O(|C|\kappa)$ bits.

6. **Masking Input Wire Values.** In this step, the communication cost is $O(1)$ times the size of inputs to the inner protocols, i.e. the size of the committed inputs $O(|C| + \mathsf{poly}(n, \kappa))$ bits.

7. **Ouputting Output Masks.** This step only contains local computation of the parties and requires no communication. The output size of this step is $O(W_O^{V_j})$ bits for each virtual server V_j, resulting in a total output size of $O(|C|)$ bits.

8. **Outputting Input Labels.** This step only contains local computation of the parties and requires no communication. for each input wire of a virtual server's local circuit whose value does not come from reconstructions, the virtual servers output $2ck = O(N)$ ciphertexts of $O(\kappa)$ bits. The total output size is $O(|C|\kappa + \mathsf{poly}(n, \kappa))$ bits.

9. **Outputting Garbled Circuits.** This step only contains local computation of the parties and requires no communication. For each AND gate and output wire of each virtual server's local circuit, the parties output ciphertexts of total size $O(\kappa)$. The total output size is $O(|C|\kappa + \mathsf{poly}(n, \kappa))$ bits.

In addition, the messages of each round in Π_{in} should be committed via AVID. Since Π_{in} is constant-round, the number of invocations of $\mathcal{F}_{\mathsf{AVID}}$ is $O(N)$, the total message size of these instances of AVID is just the communication cost of the inner protocols, i.e. $O(|C|\kappa + \mathsf{poly}(n, \kappa))$ bits. Moreover, the outputs of the inner protocols should also be committed. This can be done with $O(N)$ instances of $\mathcal{F}_{\mathsf{AVID}}$ of total message size of $O(|C|\kappa + \mathsf{poly}(n, \kappa))$ bits. Thus, the total communication cost of the garbling phase is $O(|C|\kappa + \mathsf{poly}(n, \kappa))$ bits.

Verification Phase. We analyze the communication cost of the verification phase step by step as follows:

1. **Verification of the Sharings.** The communication cost of this step is independent of the circuit size, i.e. $\mathsf{poly}(n, \kappa)$ bits.

2. **Verification of Local Computation.** The invocations of $\mathcal{F}_{\mathsf{ABA}}$ are independent of the circuit size. The communication of the retrieving process of each $\mathsf{VS}^{P_{j,i}}$ is also independent of the circuit size. The cost of the retrieving process of each $\mathsf{Sh}_\alpha^{P_{j,\beta}}$, $\mathsf{IS}^{P_{j,\beta}}$, $\mathsf{MS}^{P_{j,\beta}}$, and $\mathsf{OS}^{P_{j,\beta}}$ is $O(n)$ times the size of the set plus $O(\mathsf{poly}(n, \kappa))$ bits caused by AVID. The total size of these sets is $O(|C|\kappa + \mathsf{poly}(n, \kappa))$, resulting in a total communication cost of $O(|C|n\kappa + \mathsf{poly}(n, \kappa))$. The communication cost of the OEC process of seeds is independent of the circuit size. The final retrieving process of the output sets requires communication of $O(|C|n\kappa + \mathsf{poly}(n, \kappa))$ bits like the retrieving process of the output sets of checked virtual servers. As analyzed above, the total communication cost in this step is $O(|C|n\kappa + \mathsf{poly}(n, \kappa))$ bits.

As we analyzed above, the total communication cost of the verification phase is $O(|C|n\kappa + \mathsf{poly}(n, \kappa))$ bits.

Evaluation Phase. This phase only contains local computation of the parties and requires no communication.

As analyzed above, the total communication cost of Π is $O(|C|n\kappa + \mathsf{poly}(n, \kappa))$ bits.

J.2 Analysis of Round Complexity

Setup Phase. The setup phase can be computed in constant rounds by a general AMPC (for example using [CGHZ16]). We use R_{Setup} to denote the number of rounds required to compute $\mathcal{F}_{\mathsf{Setup}}$.

Sharing Phase. In the sharing phase, all the shares dispersed via AVID can be dispersed in parallel in 6 rounds, and the retrieving process of the shares requires 1 extra round. The dispersal and retrieving of each commitment of $\mathsf{Sh}_i^{P_{j,\alpha}}$ can be done in parallel with the shares. Then, the parties invoke the ACS functionality $\mathcal{F}_{\mathsf{ACS}}$. The parties then commit $\mathsf{IS}^{P_{j,i}}$, the dispersal process of the commitments requires 6 rounds. Thus, the total number of rounds required in the sharing phase is 13, where $\mathcal{F}_{\mathsf{ACS}}$ is invoked one time with parameter $(n, n-t)$.

Local Computation Phase. This phase only contains local computation.

Garbling phase. In the garbling phase, the parties emulating each virtual server need to jointly compute two layers of multiplications of additive sharings for each AND gate and one layer of multiplications for each output gate. The servers also need to open $\lambda_w \oplus v_w$ for each input wire w of the local circuit. The multiplications for all the gates and the opening of input wire values can be performed in parallel. Each multiplication only requires one round, and the opening of an additive sharing also requires one round, so the inner protocol requires 2 rounds. In each round, the message sets are committed in 6 rounds. The dispersal process can be done in parallel with sending the messages in the current round. After the final round, the parties should commit the output sets, which requires another 6 rounds. Thus, the garbling phase requires 18 rounds in total.

Verification Phase. In the verification phase, the servers first need to agree on a random coin. This can be done by preparing a degree-t Shamir sharing of the coin and then reconstructing the secret to all the parties. The Shamir sharing can be prepared with ACSS in parallel with the setup phase. The reconstruction can be done in parallel with the garbling phase. Then, each set $\mathsf{VS}^{P_{j,i}}$ needs to be committed. This requires 6 rounds, but it can be performed in parallel with the garbling phase. The verification of local computation must be performed after the garbling phase terminates. To get the set Ter, the parties invoke $\mathcal{F}_{\mathsf{ACS}}$. Then the reconstructions of coins to determine whether each virtual server in Ter needs to be checked require 1 extra round. Then, the opening of the commitments and the reconstruction of the seeds require 1 extra round. Finally, after the verification, the opening of output sets requires another round. Thus, the verification phase needs 3 extra rounds plus an invocation of $\mathcal{F}_{\mathsf{ACS}}$ with parameter $(N, N/4)$.

Evaluation Phase. This phase only contains local computation.

As analyzed above, the main protocol Π requires $34 + R_{\mathsf{Setup}}$ rounds plus two invocations of $\mathcal{F}_{\mathsf{ACS}}$, where R_{Setup} is a constant. One additional invocation of $\mathcal{F}_{\mathsf{ACS}}$ is required for instantiating $\mathcal{F}_{\mathsf{Setup}}$. In particular, the AMPC protocol [CGHZ16] is also constant-round in the $\mathcal{F}_{\mathsf{ACS}}$-hybrid model, and it only requires 1 invocation of $\mathcal{F}_{\mathsf{ACS}}$. Using the protocol [CGHZ16] to instantiate $\mathcal{F}_{\mathsf{Setup}}$, we obtain a constant-round AMPC protocol in the $\mathcal{F}_{\mathsf{ACS}}$ model, where $\mathcal{F}_{\mathsf{ACS}}$ is invoked 3 times sequentially.