Towards Building Scalable Constant-Round MPC from Minimal Assumptions via Round Collapsing

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

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

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

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

Abstract

In this work, we study the communication complexity of constant-round secure multiparty computation (MPC) against a fully malicious adversary and consider both the honest majority setting and the dishonest majority setting. In the (strong) honest majority setting (where $t = (1/2 - \epsilon)n$ for a constant ϵ), the best-known result without relying on FHE is given by Beck et al. (CCS 2023) based on the LPN assumption that achieves $O(|C|\kappa)$ communication, where κ is the security parameter and the achieved communication complexity is independent of the number of participants. In the dishonest majority setting, the best-known result is achieved by Goyal et al. (ASIACRYPT 2024), which requires $O(|C|n\kappa)$ bits of communication and is based on the DDH and LPN assumptions.

In this work, we achieve the following results: (1) For any constant $\epsilon < 1$, we give the first constant-round MPC in the dishonest majority setting for corruption threshold $t < (1-\epsilon)n$ with $O(|C|\kappa + D(n+\kappa)^2\kappa + n^3)$ communication assuming random oracles and oblivious transfers, where D is the circuit depth. (2) We give the first constant-round MPC in the standard honest majority setting (where $t = (n-1)/2$) with $O(|C|\kappa + D(n+\kappa)^2\kappa + n^3)$ communication only assuming random oracles.

Unlike most of the previous constructions of constant-round MPCs that are based on multiparty garbling, we achieve our result by letting each party garble his local computation in a non-constant-round MPC that meets certain requirements. We first design a constant-round MPC that achieves $O(|C|\kappa + Dn^2\kappa)$ communication assuming random oracles in the strong honest majority setting of $t = n/4$. Then, we combine the party virtualization technique and the idea of MPC-in-the-head to boost the corruption threshold to $t < (1-\epsilon)n$ for any constant $\epsilon < 1$ assuming oblivious transfers to achieve our first result. Finally, our second result is obtained by instantiating oblivious transfers using a general honest-majority MPC and the OT extension technique built on random oracles.

Contents

1 Introduction **4**
- 1.1 Our Contribution . . . 4
- 1.2 Related Works . . . 5

2 Technical Overview **6**
- 2.1 Compiling an SS-Based Protocol to a Constant-Round Protocol via Black-Box Garbling . . . 7
- 2.2 Towards Constant Communication . . . 10
- 2.3 Party Virtualization . . . 10

3 Preliminaries **11**
- 3.1 Linear Secret Sharing Schemes . . . 11
- 3.2 Garbling Scheme . . . 13
- 3.3 Symmetric Key Encryption Scheme . . . 14
- 3.4 Chernoff Bound . . . 15

4 Constant-Round MPC from Black-Box Garbling **15**
- 4.1 Abstract Non-Constant-Round MPC Protocol . . . 15
- 4.2 Towards Constant-Round MPC with Malicious Security . . . 16

5 Boosting the Efficiency via Concrete Garbling Schemes **18**
- 5.1 The Choice of the Garbling Scheme . . . 18
- 5.2 Boosting the Efficiency . . . 18

6 Instantiation of the Abstract Protocol for SIMD Circuits **22**

7 Overview of the Dishonest Majority Constant-Round MPC **24**

A The Security Model **33**

B Basic Algebraic Geometry **33**

C Security Proof for Protocol Π_1 **34**

D Security Proof for Protocol Π_1' **39**

E Cost Analysis for Π_1' **47**

F Instantiation of the Abstract Protocol **48**
- F.1 Handling Network Routing. . . . 48
- F.2 Instantiation of Linear Secret Sharing Scheme. . . . 50
- F.3 Instantiation of the Protocol . . . 52
  - F.3.1 Functionalities for Preprocessing and Input. . . . 52
  - F.3.2 Subprotocols. . . . 53
  - F.3.3 Main Protocol. . . . 55
- F.4 Cost Analysis for Π_0 . . . 56

G Proof of Theorem 6 **57**

H Realizing the Functionalities **63**
- H.1 Realizing $\mathcal{F}_{\mathsf{prep}}$ . . . 63
- H.2 Realizing $\mathcal{F}_{\mathsf{input}}$ . . . 71

I Analysis of Rounds for Π_1' **74**

J Dishonest Majority Constant-Round MPC **74**
J.1 Subprotocols . . . 74
J.2 Protocol Description . . . 76
J.3 Security Proof . . . 83
J.4 Cost Analysis for Π_2 . . . 103
J.4.1 Analysis of Communication Complexity . . . 103
J.4.2 Analysis of Rounds . . . 105

K Corollary in the Standard Honest Majority Setting **106**

1 Introduction

Secure multiparty computation (MPC) [Yao82, GMW87] enables a set of parties to evaluate a public function on their private inputs. By now, many improvements have been made to the efficiency of MPC protocols from different aspects, such as computation complexity, communication complexity, and round complexity. However, existing protocols remain inefficient when running among a large number of parties that are geographically far away from each other. To fit this situation, we need *round-efficient* and *scalable* protocols. The two main techniques of constructing MPC protocols, garbled circuits and secret sharings, have been used in constructing efficient protocols in round complexity and communication complexity respectively. However, these protocols often have poor performance in the other criteria.

MPC from Secret Sharings. The communication cost of secret-sharing-based MPC protocols has achieved good scalability in the number of parties. In particular, linear communication has been achieved for a long time by the well-known DN protocol [DN07] for semi-honest security and [BFO12] for malicious security with honest majority, and SPDZ protocol [DPSZ12] with dishonest majority. Moreover, recent works have even achieved an $O(|C|)$ total communication to compute a circuit C, which is independent of the number of parties, either in the strong honest majority setting where $t = (1/2-\epsilon)n$ for any constant $\epsilon < 1/2$ [GPS21] or in the dishonest majority setting where $t = (1-\epsilon)n$ for any constant $\epsilon < 1$ in the preprocessing model [GPS22], even with information-theoretic security. However, MPC protocols based on secret sharings are usually constructed in a gate-by-gate fashion, and the round complexity of such protocols grows linearly in the depth of the circuit. When the network latency between the parties is large, these protocols become quite inefficient in computing deep circuits.

MPC from Garbled Circuits. To achieve constant-round MPCs, the most common approach is based on garbled circuits. Protocols from garbled circuits can overcome the drawback of the large round complexity of those protocols based on secret sharings. In particular, protocols with constant rounds have been constructed since the famous BMR protocol [BMR90] and its followup work [DI05]. Up to now, a long line of works have improved the efficiency of constant-round MPC protocols in the BMR framework. However, this kind of protocols still required $O(|C|n^2\kappa)$ bits of communication for a long time, which is inefficient when running among a large party set.

Recently, the $O(|C|n^2\kappa)$ barrier has been overcome in both the (strong) honest majority setting and the dishonest majority setting. The significant progress by Beck et al. [BGH+23] achieves $O(|C|\kappa)$ communication complexity for constant-round MPC in the strong honest majority setting of $(1/2-\epsilon)n$ corruption under the assumption of LPN, and the best achieved communication complexity in the dishonest majority setting is $O(|C|n\kappa)$ [GLM+25] based on the DDH and LPN assumptions. However, both constructions require heavy cryptographic primitives that incur a large computation overhead, whereas non-constant-round MPCs are mostly based on information-theoretic tools.

Thus, a natural question is whether we can build MPC protocols that have the advantage of both approaches. In particular, we ask the following question:

"Can we construct an MPC protocol that achieves both constant round complexity and constant communication complexity (in the number of parties) from lightweight cryptographic primitives?"

1.1 Our Contribution

In this work, we answer the above question affirmatively and obtain constant-round MPC protocols with constant communication complexity in various settings only assuming random oracles and oblivious transfers (OTs). Our first result targets for the strong honest majority setting against $t = n/4$ corrupted parties.

Theorem 1. *Assuming random oracles, there exists a computationally secure 5-round MPC protocol against a fully malicious adversary controlling up to $n/4$ parties with communication of $O(|C|\kappa + Dn^2\kappa)$, where $|C|$ is the circuit size, D is the circuit depth, and κ is the computational security parameter.*

When relying on the party virtualization technique [Bra87] and OTs, we can further extend our first result to the dishonest majority setting where $t = (1-\epsilon)n$ for any constant $0 < \epsilon < 1$.

Theorem 2. *Assuming random oracles and random OTs, for any constant $0 < \epsilon < 1$, there exists a computationally secure $(12 + R^{\mathsf{ROT}})$-round MPC protocol against a fully malicious adversary controlling up to $(1-\epsilon)n$ parties with communication of $O(|C|\kappa + D(n+\kappa)^2\kappa + n^3)$ bits plus $O(|C| + D(n+\kappa)^2)$ instances of ROT of message length $O(\kappa)$, where $|C|$ is the circuit size, D is the circuit depth, R^{ROT} is the number of rounds for an instance of ROT, and κ is the computational security parameter.*

Finally, we note that in the standard honest majority setting, oblivious transfers can be instantiated by a general honest-majority MPC protocol and the OT extension technique [IKNP03], thus obtaining the following corollary.

Corollary 1. *Assuming random oracles, there exists a computationally secure 28-round MPC protocol against a fully malicious adversary controlling up to $(n-1)/2$ parties with communication of $O(|C|\kappa + D(n+\kappa)^2\kappa + n^3)$ bits, where $|C|$ is the circuit size, D is the circuit depth, and κ is the computational security parameter.*

To achieve our result, our main technique is a compiler that converts any non-constant-round MPC protocol that meets certain requirements to a constant-round MPC protocol, and the compiler can be based on any (projective) garbling scheme in a black-box way. Our first result only achieves security against 1/4 corruption. To boost the corruption threshold, we use the party virtualization technique [Bra87], together with the idea of MPC-in-the-head to achieve malicious security. We refer the readers to Section 2 for an overview of our techniques.

Limitations of Our Results. Our results suffer from the following two limitations. First, the communication complexity we achieve contains a term that depends on the circuit depth, which usually appears in non-constant-round MPC such as [GPS21, GPS22, EGP+23] but not in constant-round MPC constructions [BGH+23, GLM+25]. This term comes from our instantiation of the underlying non-constant round MPC protocol, which requires $O(|C| + Dn^2)$ communication due to the use of packed secret sharing. Thus, we only achieve $O(|C|\kappa)$ communication for circuits whose depth $D = O(|C|/(n+\kappa)^2)$.

Second, our results mainly target for *asymptotic* communication efficiency and are not ready to be used in practice. We leave the question of removing the depth-dependent term and improving the concrete efficiency of our construction to future work.

1.2 Related Works

Following the well-known BMR protocols [BMR90, DI05], a rich line of works have improved the concrete efficiency of constant-round MPC protocols in the BMR framework [LPSY15, BLO16, BLO17, WRK17, HSS17, HOSS18a, HOSS18b, BCO+21, BGH+23, GLM+25]. We compare our results with the most recent two works [BGH+23, GLM+25].

For the computation of a circuit C of size $|C|$, the communication complexity we achieve is $O(|C| \cdot \kappa)$ (here we omit the terms that are sublinear in the circuit size) in the dishonest majority setting with a constant gap. This result even improves the state-of-the-art non-constant round MPC protocol in this setting [EGP+23]. On the other hand, the state-of-the-art result by Beck et al. [BGH+23] only achieves a similar result assuming a strong honest majority. Compared with the best result in the dishonest majority setting by Goyal et al. [GLM+25], our construction improves the communication complexity by a factor of $O(n)$ when there is a constant gap in the corruption threshold.

Our constructions are based on the minimal assumption (i.e., OTs in the dishonest majority setting) plus random oracles, while the previous protocols [BGH+23, GLM+25] rely on stronger cryptographic assumptions such as DDH and LPN.

Regarding the round complexity, [BGH+23] requires at least 31 rounds and [GLM+25] requires at least 36 rounds plus the round complexity of instantiating VOLEs and OLEs. In comparison, our construction

requires $12 + R^{\mathsf{ROT}}$ rounds in the dishonest majority setting, and 28 rounds in the honest majority setting, which are more efficient than [BGH+23, GLM+25].

We note that a concurrent work [HKN+25] also focuses on the dishonest majority setting against $t = (1-\epsilon)n$ corruption, assuming OT and random oracles. Their construction achieves $O(|C|n\kappa)$ communication and is more efficient for small circuits. In comparison, our result achieves $O(|C|\kappa + D(n+\kappa)^2\kappa + n^3)$ communication, which improves [HKN+25] by a factor of $O(n)$. From the technique side, [HKN+25] follows the BMR protocol [BMR90] and [WRK17], and makes use of packed secret sharing to shave a factor of $O(n)$ in the communication complexity. Our approach deviates from [HKN+25] by constructing a compiler that converts any non-constant-round MPC protocol that meets certain requirements to a constant-round MPC protocol.

Other Related Works. Another line of works including [GS18, BL18, ACGJ19, ABT19, ACGJ20] target for the round-optimal MPC in various settings whereas our work mainly focuses on improving the communication complexity as long as the protocol achieves a constant number of rounds. Based on the black-box use of FHE, PCP and PCPP machinery [CMOS25] achieve an 11-round, maliciously secure 2PC protocol, with communication complexity that is independent of the circuit size and depends only on the size of input/output times the security parameter. However, their scheme uses very strong assumptions (circular-secure FHE with bootstrapping) and is far from being practically efficient.

We note that [GS18, BL18] and our work share a similar starting idea of using garbled circuits to do round collapsing. But except for the starting idea, our technique is very different from that in [GS18, BL18]. The construction in [GS18, BL18] works for any non-constant-round MPC only assuming OTs. Instead, our work only focuses on non-constant-round MPC that meets certain requirements and assumes random oracles in addition. On the other hand, the communication complexity of [GS18, BL18] is at least $\Theta(|C|n^2\kappa)$ since each party needs to maintain the state of all parties and generates a pair of wire labels for each bit in the state. Note that this even does not count the cost of the underlying non-constant-round MPC protocol. In comparison, our construction achieves $O(|C|\kappa)$ communication.

It worth mentioning that [ACGJ20] also achieves $\tilde{O}(|C|)$ communication in the strong honest majority setting where the $\tilde{O}$ notation suppresses polynomial factors in the security parameter κ and logarithmic polylog factors in the number of parties n. Very informally, their result is achieved by choosing a random committee at the beginning of the computation such that at least one party in the committee is honest, and then running a constant-round dishonest majority MPC among parties in the committee only. Note that to ensure that the random committee contains at least one honest party with overwhelming probability, the committee size should be proportional to the security parameter κ. We estimate that the construction in [ACGJ20] requires at least $\Theta(|C|\kappa^7)$ communication. On the other hand, our work only requires $O(|C|\kappa + D(n+\kappa)^2\kappa)$ communication.

From the technique side, the use of a random committee in [ACGJ20] makes their construction only secure against a *static* adversary. Indeed, an adaptive adversary, after learning the choice of the committee, can just corrupt parties in the committee and easily break down the security. Although our construction makes use of the party virtualization technique [Bra87] to boost the corruption threshold, where each virtual party is also simulated by a random committee (of constant size), our construction can potentially achieve the adaptive security. As we analyzed in Section 7, an adversary, even choosing corrupted parties after learning the choice of all committees for virtual parties, cannot break the security of our construction.

2 Technical Overview

We give a high-level overview of the main techniques used in this paper. We focus on constructing constant-round MPC protocols for a Boolean circuit C consisting of AND and XOR gates.

2.1 Compiling an SS-Based Protocol to a Constant-Round Protocol via Black-Box Garbling

Starting Idea and Technical Difficulty in Our Setting. At a high level, we start with a multi-round protocol Π and attempt to collapse the number of rounds using a compiler. For simplicity, we assume the entire protocol Π is running over a broadcast channel. Our starting point is the following idea: For each round in the multi-round protocol Π, each party P_i would create a garbled circuit corresponding to the next-message function of the protocol. Thus, if the number of rounds in Π is ρ, the total number of garbled circuits would be $n \cdot \rho$. All of these garbled circuits are then sent to an evaluator (say P_1). The evaluator will attempt to execute these garbled circuits non-interactively.

To be more concrete, after receiving garbled circuits from all parties, the evaluator emulates the underlying multi-round MPC protocol as follows:

1. The evaluator first obtains from all parties the input wire labels for garbled circuits corresponding to the next-message functions in the first round. Then he evaluates all parties' garbled circuits for the first round and obtains the output wire labels, which are used as the input wire labels for the next-message functions in the second round.

2. Following the above, with input wire labels for the j-th round, the evaluator computes all parties' garbled circuits for the j-th round and obtains the input wire labels for the $(j+1)$-th round.

3. Finally, the evaluator obtains the protocol output.

However, to make the above idea work, for each bit z sent from a party P_i to a party P_j in the underlying MPC protocol, P_i and P_j have to use the same wire labels for z when computing their garbled circuits. Only in this way, after the evaluator computes P_i's garbled circuit and obtains P_i's wire label for z, he can use it as the input wire label for z to compute P_j's garbled circuit in the next round. Unfortunately, this is not secure when P_i colludes with the evaluator (or even P_i is just the evaluator) since the evaluator would learn from P_i the wire labels corresponding to both $z = 0$ and $z = 1$. This allows the evaluator to try both $z = 0$ and $z = 1$ by using the corresponding wire label in P_j's garbled circuit, which breaks the security of the underlying MPC protocol.

Therefore, for security reasons, P_i and P_j cannot use the same wire labels for z. But then, it is not clear how the evaluators could obtain the input wire labels for P_j's garbled circuits. In the literature, the same starting idea has been used in various forms in prior works. In the context of one-time programs [GKR08, GIS+10], these wire labels are made available using hardware tokens. In [GS18], this difficulty is bypassed by cleverly combining OT protocols with garbled circuits, while in [BL18], the authors use witness encryption to "commit " the inputs to the garbled circuit to overcome this difficulty. However, both constructions in [GS18, BL18] require every party to keep track of the masked state of all parties. In particular, each party will generate a pair of wire labels for every bit in the masked state of all parties. This results in at least $\Theta(|C|n^2\kappa)$ communication for computing a function of size $|C|$.

In our setting, we want the output of the garbled circuits of round j to allow the evaluator to compute wire labels to be used in round $j+1$ directly, without resorting to hardware tokens or OT protocols.

To this end, at the very beginning of the protocol, each party P_i secret-shares all the input wire labels for all the garbled circuits prepared by P_i. Then when a party prepares the garbled circuits, the garbled circuits would have the received wire label shares hardcoded. What we want to achieve in the following is to let all parties' garbled circuits of round j, in addition to computing their original outputs, compute a secret sharing of each input wire label in round $j+1$ as well. This would allow the evaluator to reconstruct the wire labels to be used in round $j+1$ directly from the output of the garbled circuits of round j.

We first consider a simple scenario: Suppose that each garbled circuit in round j, in addition to being able to compute its own output, could also compute the output of every other garbled circuit in round j. In that case, the solution is easy:

- Each garbled circuit in round j knows the full input of each garbled circuit in round $j+1$. Thus, it can output the shares of the relevant wire labels for round $j+1$ and the computation can continue.

Note that this idea naturally resists the attack mentioned above. This is because what a malicious evaluator can learn from corrupted parties are just their shares, which reveal no information about the wire labels. On the other hand, from honest parties' shares, the evaluator can only reconstruct a single wire label for each input wire corresponding to the actual input for garbled circuits in round $j+1$.

Of course, the main difficulty with this approach is that a garbled circuit prepared by P_i for round j can only compute the message of P_i in round j, not of other parties.

Protocols with Reconstruction Only Messages. We do not know how to fix the above problem in general. However, we note that many protocols in the literature have a special property which we call *Reconstruction Only Messages (ROM)*. Very roughly, the only messages sent by the parties in the protocol (after a constant-round preprocessing phase) are shares of a message z and the parties are trying to reconstruct z publicly. In more details:

- Parties only need to do public reconstruction in the online phase (that is to say the online phase can be run over the broadcast channel and there is no private message);
- And parties only use the reconstruction results, *not the individual shares*, as input for their next-message functions.

For example, consider the well-known DN protocol [DN07]. The idea is to let all parties compute a degree-t Shamir secret sharing of each wire value in the circuit. In the following, we use $[x]_t$ to denote a degree-t Shamir secret sharing of the secret x. Then linear gates can be handled locally due to the linear homomorphism of the Shamir secret sharing scheme. For a multiplication gate with input sharings $[x]_t, [y]_t$, all parties first prepare a pair of random sharings in the form of $([r]_t, [r]_{2t})$ in the preprocessing phase. Then in the online phase, all parties locally compute $[z]_{2t} = [x]_t \cdot [y]_t + [r]_{2t}$. To obtain $[x \cdot y]_t$, all parties together reconstruct the secret z and compute $[x \cdot y]_t = z - [r]_t$. Thus, the goal is to let all parties learn the reconstruction result z which is used in their next-message functions.

Round Collapsing for Protocols with Reconstruction Only Messages. Now we will try to compile a protocol satisfying ROM using our round collapsing compiler. To better explain our idea, we take the DN protocol as an example in the following description.

Following our idea, at the very beginning of the protocol, each party P_i secret-shares all the input wire labels for all the garbled circuits prepared by P_i. From the property of ROM, every input wire of P_i's garbled circuits coming from the online phase carries the secret z of a degree-$2t$ Shamir secret sharing $[z]_{2t}$ that should be reconstructed to P_i. Suppose P_i chooses $(k_{w,0}, k_{w,1})$ as the wire labels associated with the input wire w that carries the input value z. Then P_i will shares $[k_{w,0}]_t, [k_{w,1}]_t$ at the beginning of the protocol.

We note that if all the garbled circuits knew what the message z being reconstructed is, the solution would be easy as well:

- Each garbled circuit of P_j just outputs P_j's share of $[k_{w,z}]_t$ (for every input wire w of garbled circuits for the next round).

Following a similar argument, this solution also resists the attack we mentioned above: A malicious evaluator, even colluding with up to t corrupted parties, can only learn the wire label $k_{w,z}$ associated with the correct secret z. But again, the remaining challenge is that parties only have shares of z rather than z itself.

We note that from $z, k_{w,0}, k_{w,1}$, the wire label $k_{w,z}$ can be computed by the following equation

$$k_{w,z} = k_{w,0} + z \cdot (k_{w,1} - k_{w,0}).$$

Since the Shamir secret sharing is multiplicative, from shares of $[z]_{2t}, [k_{w,0}]_t, [k_{w,1}]_t$, all parties can compute a Shamir secret sharing of $k_{w,z}$ from local computations. To be more concrete, we modify the way how each party P_i shares his input wire labels: For an input wire w with chosen labels $(k_{w,0}, k_{w,1})$, P_i shares $[k_{w,0}]_{3t}$

and $[k_{w,1} - k_{w,0}]_t$ (here we assume the corruption threshold $t < n/3$) to all parties. Recall that in the DN protocol, all parties would reconstruct a degree-$2t$ Shamir sharing $[z]_{2t}$ to P_i, and z is used as the input for the wire w in P_i's next-message function. Relying on the multiplicative property of the Shamir secret sharing scheme, all parties can locally compute

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

Thus, we ask each party P_j to garble the following computation: It takes as input P_j's view and P_j's shares of $[k_{w,0}]_{3t}$ and $[k_{w,1} - k_{w,0}]_t$. Then it computes P_j's next-message function and obtains P_j's share of $[z]_{2t}$. After that it computes P_j's share of $[k_{w,z}]_{3t}$ following Equation 1. The output is set to be P_j's shares of $[z]_{2t}$ and $[k_{w,z}]_{3t}$.

Now by evaluating all parties' garbled circuits, the evaluator can obtain the whole sharings $[z]_{2t}$ and $[k_{w,z}]_{3t}$ as output. This allows the evaluator to reconstruct z and $k_{w,z}$, and use them to evaluate P_i's next-round garbled circuit.

As for security, we still need to show that the evaluator cannot learn the other wire label $k_{w,1-z}$ in the above process. Note that the evaluator learns (1) the whole sharing of $[k_{w,z}]_{3t}$, (2) the whole sharing of $[z]_{2t}$, and (3) corrupted parties' shares of $[k_{w,0}]_{3t}$ and $[k_{w,1} - k_{w,0}]_t$. Now for any assignment of $k_{w,1-z}$, which defines $k_{w,1} - k_{w,0}$ given $[k_{w,z}]_{3t}$ and $[z]_{2t}$, there is a valid degree-t Shamir sharing $[k_{w,1} - k_{w,0}]_t$ given the shares of corrupted parties, and then a valid degree-$3t$ Shamir sharing $[k_{w,0}]_{3t} = [k_{w,z}]_{3t} - [z]_{2t} \cdot [k_{w,1} - k_{w,0}]_t$ given the shares of corrupted parties. This implies that the evaluator does not learn any information about the other wire label $k_{w,1-z}$ and the security holds.

Summary of Our Compiler for DN Protocol. We summarize our construction idea that converts the DN protocol to a constant-round MPC protocol as follows:

1. All parties run the preprocessing phase and the input phase of the DN protocol.
2. Each party chooses the input wire labels associated with the circuits of his next-message functions.
3. For every $[z]_{2t}$ that should be reconstructed to a receiver R in the DN protocol, suppose R chooses $(k_{w,0}, k_{w,1})$ as the wire labels for this wire. R distributes $[k_{w,0}]_{3t}$ and $[k_{w,1} - k_{w,0}]_t$ to all parties.
4. Each party P_j locally garbles the following computation for each reconstruction of $[z]_{2t}$: It takes as input P_j's view and P_j's shares of $[k_{w,0}]_{3t}$ and $[k_{w,1} - k_{w,0}]_t$. Then it computes P_j's next-message function and obtains P_j's share of $[z]_{2t}$. After that, it computes P_j's share of $[k_{w,z}]_{3t}$ following Equation 1. The output is set to be P_j's shares of $[z]_{2t}$ and $[k_{w,z}]_{3t}$.
5. All parties send their garbled circuits to the evaluator as well as the input wire labels associated with their preprocessing data and input data.
6. The evaluator emulates each party's next-message functions by evaluating his garbled circuits round by round. For each $[z]_{2t}$ reconstructed towards the receiver R, the evaluator obtains the whole sharings $[z]_{2t}$ and $[k_{w,z}]_{3t}$, and reconstructs the wire label $k_{w,z}$ for z, which allows the evaluator to compute R's garbled circuit in the next round. After obtaining the protocol output, the evaluator sends the output to all parties.

Achieving Malicious Security. As we have shown above, for semi-honest security, the above construction ensures that for each honest receiver R, a semi-honest evaluator, even colluding with up to t parties, can only learn the wire label $k_{w,z}$ associated with the correct secret z, but learns no information about the other wire label $k_{w,1-z}$. This forces the evaluator to honestly emulate honest parties' next-message functions in the underlying non-constant-round MPC.

We note that the above construction has almost achieved malicious security. This is because a malicious evaluator is not much stronger than a semi-honest evaluator: the only difference is that a malicious evaluator can send an incorrect protocol output to all parties, which can be detected relying on message authentication

codes. Thus to achieve malicious security, we only need to use maliciously secure preprocessing protocol and input protocol in Step 1 and ensure that parties distribute valid Shamir sharings in Step 3. After that, honest parties will just compute the garbled circuits locally and send them to the evaluator, which is not affected by the malicious behaviors of corrupted parties. (Note that when the evaluator is honest, corrupted parties may send incorrect garbled circuits to the evaluator. However, this case can be reduced to a malicious evaluator who follows the protocol and uses the incorrect garbled circuits from corrupted parties.)

In general, we only need the underlying non-constant-round MPC protocol to achieve malicious security in the preprocessing phase and the input phase, but *fail-stop* security in the online phase. We refer the readers to Section 4 for more details.

2.2 Towards Constant Communication

Now, we analyze the communication cost of the MPC construction from the above idea. First note that in the original DN protocol, the reconstruction of $[z]_{2t}$ is done by a single party P_{king} and then P_{king} distributes z to all parties. Only in this way, the DN protocol achieves a linear communication complexity per gate. Unfortunately, our above solution does not support this style of reconstruction. Even if it supports, we still need a linear communication complexity per gate while our goal is to achieve constant communication per gate. Besides, since we can only garble a binary circuit or otherwise, the secret of $[k_{w,0}]_{3t} + [z]_{2t} \cdot [k_{w,1} - k_{w,0}]_t$ will not correspond to a valid wire label, we have to use a binary extension field of size $\Omega(n)$ to be able to use the Shamir secret sharing scheme, which leads to a factor of $O(\log n)$ overhead.

Thus, to achieve constant communication, we replace the Shamir secret sharing scheme with a general multiplicative linear secret sharing scheme based on AG codes [CC06] and modify the communication-efficient non-constant-round MPC protocols [GPS21, GPS22] to work with general multiplicative linear secret sharing schemes and only require reconstructions in the online phase. The obtained non-constant-round MPC protocol is resilient to $t < n/4$ corruptions.

Another factor that affects the communication of our constant-round protocol is the size of the garbled circuits, which is determined by the complexity of all parties' *next-message functions* in the non-constant-round MPC. To obtain constant communication, we need the complexity of all parties' next-message functions to be constant in the circuit size. Unfortunately, our instantiation of the non-constant-round MPC protocol, as well as the MPC protocols in [GPS21, GPS22], while achieving constant *communication* complexity, requires linear *computation* complexity. However, we observe that most of the computations in our non-constant-round MPC protocol are addition operations while the *total* number of multiplication operations done by all parties is bounded by $O(|C|)$. To overcome the effect of linear computations, we utilize the freeXOR technique [KS08] in the local garbling process (see Section 5 for more details) assuming random oracles. In this way, the garbled circuits' size only depends on the number of multiplication operations done by all parties, which is constant per gate.

With the above two improvements, we obtain our first construction against $1/4$ corruption with constant communication in the number of parties.

2.3 Party Virtualization

So far, the above construction only works when the corruption threshold is bounded by $t = n/4$. To relax this restriction, we use the party virtualization technique [Bra87] to boost the corruption threshold. At a high level, each time we randomly select a constant number of parties to form a committee and let parties in this committee emulate a virtual party in our first protocol. We show that for any constant $0 < \epsilon < 1$, there exists a constant c such that when the number of virtual parties $N = O(n + \kappa)$, with overwhelming probability, for any $t = (1 - \epsilon)n$ corrupted parties, at least $3/4$ fraction of virtual parties are simulated by a committee that contains at least a single honest party. Then, it is sufficient to let parties in each committee run a generic dishonest majority MPC protocol to emulate the computation of the corresponding virtual party in our first protocol.

In our first protocol, each party locally garbles his next-message functions. Thus, after applying the party virtualization technique, we let parties in each committee generate the garbled circuits of the corresponding

virtual party utilizing the multiparty garbling technique [BMR90, DI05] which is secure even when $c-1$ parties of the c parties in the committee are dishonest. Note that since the committee size is a constant c, we only need the underlying multiparty garbling protocol to achieve a cost that grows linearly with the circuit size (but can be polynomial in the number of parties).

To achieve malicious security, a direct way is to apply a maliciously secure MPC to do the multiparty garbling. However, these protocols either require non-symmetric cryptographic assumptions such as linearly homomorphic encryption [BDOZ11] and LPN [RS22] or lead to a non-constant multiplication overhead on the communication complexity [LOS14, KOS16]. Our idea is to use a modified version of the "Watchlist" technique based on MPC-in-the-head [IKOS07, IPS08, IKSS22, FR23].

Suppose the virtual parties are denoted by $V_1, \ldots, V_N$ where each virtual party V_i is simulated by a randomly selected committee. At a high level, we still use a semi-honest dishonest majority MPC protocol among parties in each committee to generate the garbled circuits of each virtual party. Then, each party P_i chooses a random subset of virtual parties and checks the generation of the garbled circuits of these virtual parties. Intuitively, if for a constant fraction of virtual parties, corrupted parties deviate from the protocol when generating the garbled circuits, it would be caught by an honest party during the verification step with overwhelming probability. Thus, if all honest parties are happy with the verification, it ensures that most ($\geq 3/4$) of the virtual parties' garbled circuits are correctly generated. We refer the readers to Section 7 for a more detailed overview of our solution.

3 Preliminaries

Notation. Let κ denote the secure parameter, and let $\mathbb{F}_q$ denote the finite field with q elements. For a matrix $\mathcal{M}$, let $\mathcal{M}_{[:,i]}$ denote its i-th column. For a list $\mathcal{L}$, let $\mathcal{L}[i]$ denote its i-th entry. 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

$$\boldsymbol{u} \otimes \boldsymbol{v} = (u_i \cdot v_j)_{i \in \{1,\ldots,k\}, j \in \{1,\ldots,\ell\}} = (u_1 v_1, \ldots, u_1 v_\ell, \ldots, u_k v_1, \ldots, u_k v_\ell)$$

for $\boldsymbol{u} = (u_1, \ldots, n_k), \boldsymbol{v} = (v_1, \ldots, v_\ell)$. We use $r \xleftarrow{\$} \mathcal{R}$ to denote that r is sampled uniformly from $\mathcal{R}$. For random variables A and B, we use $A \equiv B$ to denote that A and B have the same distribution. When $X = X_\kappa$ and $Y = Y_\kappa$ are family of distributions indexed by a security parameter κ, we say that X and Y are computationally indistinguishable, denoted $X \equiv_c Y$, if for every polynomial $t(\cdot)$, $\max_{D \in \mathcal{D}_{t(\kappa)}} \Delta_D(X, Y) = \mathsf{negl}(\kappa)$. Here $\Delta_D(X, Y)$ is the advantage of a circuit D in distinguishing X and Y, defined by

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

and $\mathcal{D}_t$ is the set of all probabilistic circuits of size t.

Security Model. In this work, we use the client-server model for secure multi-party computation. We define the security of multiparty computation in the *real and ideal world paradigm* [Can00]. Informally, we consider a protocol Π to be secure if any adversary's view in its execution in the real world can also be simulated in the ideal world. For more details, we refer the readers to Section A.

3.1 Linear Secret Sharing Schemes

In this section, we introduce the basic definitions of linear secret sharing schemes (LSSS). Then we define the property of multiplicative reconstruction on an LSSS.

Definition 1. (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 2. (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.*

Definition 3. *We say an (n, t, k, ℓ)-LSSS Σ has d-multiplicative reconstruction if there exists (n, t, k, ℓ^α)-LSSSs $\Sigma^{(\alpha)}$ for $\alpha = 2, \ldots, d$ such that:*

- *For any $\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)} \in \mathbb{F}_q^k$ and Σ-sharings $\boldsymbol{c}^{(1)} = (\boldsymbol{c}_1^{(1)}, \ldots, \boldsymbol{c}_n^{(1)})$, $\boldsymbol{c}^{(2)} = (\boldsymbol{c}_1^{(2)}, \ldots, \boldsymbol{c}_n^{(2)})$ of $\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)}$ respectively, $(\boldsymbol{c}_1^{(1)} \otimes \boldsymbol{c}_1^{(2)}, \ldots, \boldsymbol{c}_n^{(1)} \otimes \boldsymbol{c}_n^{(2)})$ is a $\Sigma^{(2)}$-sharing of $\boldsymbol{s}^{(1)} * \boldsymbol{s}^{(2)}$.*
- *For any $\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)} \in \mathbb{F}_q^k$, $\Sigma^{(j-1)}$-sharing $\boldsymbol{c}^{(1)} = (\boldsymbol{c}_1^{(1)}, \ldots, \boldsymbol{c}_n^{(1)})$ of $\boldsymbol{s}^{(1)}$, and Σ-sharing $\boldsymbol{c}^{(2)}$ of $\boldsymbol{s}^{(2)}$, $(\boldsymbol{c}_1^{(1)} \otimes \boldsymbol{c}_1^{(2)}, \ldots, \boldsymbol{c}_n^{(1)} \otimes \boldsymbol{c}_n^{(2)})$ is a $\Sigma^{(j)}$-sharing of $\boldsymbol{s}^{(1)} * \boldsymbol{s}^{(2)}$ for each $j = 3, \ldots, d$.*

For convenience, for an LSSS Σ, we write

$$\Sigma.\mathsf{Sh}(\boldsymbol{s}, \boldsymbol{r}) = (\Sigma.\mathsf{Sh}_1(\boldsymbol{s}, \boldsymbol{r}), \ldots, \Sigma.\mathsf{Sh}_n(\boldsymbol{s}, \boldsymbol{r})),$$

where $\Sigma.\mathsf{Sh}_i(\cdot,\cdot) : \mathbb{F}_q^k \times \mathbb{F}_q^{n\ell} \to \mathbb{F}_q^\ell$ for each $i = 1, \ldots, n$.

Now we state a lemma from [ZLC+08] which shows that a LSSS Σ with 3-multiplicative reconstruction is strong multiplicative, i.e. the secret of a $\Sigma^{(2)}$-sharing is uniquely determined by any $n - t$ shares.

Lemma 1. ([ZLC+08]). *If an (n, t, k, ℓ)-LSSS Σ has 3-multiplicative reconstruction with $\Sigma^{(2)}, \Sigma^{(3)}$, then the secret of a $\Sigma^{(2)}$-sharing is uniquely determined by any $n - t$ shares.*

Additionally, 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].

Remark 1. *For an (n, t, k, ℓ)-LSSS Σ, we say a set S of e shares in $\mathbb{F}_q^\ell$ are valid if they are from a valid Σ-sharing. We say a set S of e shares uniquely determines the secret $\boldsymbol{s}$ if any Σ-sharing that matches the shares in S has secret $\boldsymbol{s}$.*

Our construction requires the following algorithms. The first algorithm Alg_1 takes as input a set S of e shares and outputs $\boldsymbol{s}$ if the shares in S are valid and uniquely determine the secret $\boldsymbol{s}$, and $\perp$ otherwise. We sketch the construction of Alg_1 as follows:

1. *The algorithm first turns the sharing algorithm $\Sigma.\mathsf{Sh} : \mathbb{F}_q^k \times \mathbb{F}_q^{n\ell} \to \mathbb{F}_q^{n\ell}$ into matrix representation, i.e.*

$$\mathcal{M} \cdot (\boldsymbol{s}, \boldsymbol{r})^T = (\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n)^T.$$

Let $S = \{\boldsymbol{c}_{j_1}, \ldots, \boldsymbol{c}_{j_e}\}$. Let $\mathcal{M}_S$ denote the matrix that consists of the $j_1, \ldots, j_e$-th rows of $\mathcal{M}$. The problem is transformed into solving the equation

$$\mathcal{M}_S \cdot (\boldsymbol{s}, \boldsymbol{r})^T = (\boldsymbol{c}_{j_1}, \ldots, \boldsymbol{c}_{j_e})^T.$$

2. *The algorithm uses the Gaussian elimination to obtain a linear space for all possible solutions within polynomial time. If all the solutions of $(\boldsymbol{s}, \boldsymbol{r})$ share the same $\boldsymbol{s}$, the algorithm outputs $\boldsymbol{s}$ as the reconstruction result. Otherwise, the algorithm outputs $\perp$.*

The second algorithm Alg_2 takes as input a set S of e shares and outputs a Σ-sharing that matches the shares in S if it exists, and $\perp$ otherwise. This can be easily obtained by taking an arbitrary solution of $(\boldsymbol{s}, \boldsymbol{r})$ from the solution space obtained from the process of the first algorithm and then running the sharing algorithm.

The third algorithm Alg_3 takes as input a set I of at most t indices and generates random shares for parties with indices in I. Denote the randomized algorithm of sampling the shares for a set I ($|I| \leq t$) by $\Sigma.\mathsf{Sh}_I : \perp \to (\mathbb{F}_q^\ell)^{|I|}$, which is defined by running $\Sigma.\mathsf{Sh}_I() = \pi_I(\Sigma.\mathsf{Sh}(\boldsymbol{o}, \boldsymbol{r}))$ where $\boldsymbol{o} \in \mathbb{F}_q^k$ is an all-zero vector and $\boldsymbol{r} \in \mathbb{F}_q^{n\ell}$ is a randomly sampled vector. Note that from the property of t-privacy, the shares of parties in I obtained from $\Sigma.\mathsf{Sh}_I$ distribute the same as those obtained from $\Sigma.\mathsf{Sh}(\boldsymbol{s}, \boldsymbol{r})$ for any secret $\boldsymbol{s}$ and a uniformly random vector $\boldsymbol{r} \in \mathbb{F}_q^{n\ell}$.

Finally, the last algorithm Alg_4 takes as input a set S_1 of shares of parties in I_1 and a set I_2 of indices such that $I_1 \cap I_2 = \emptyset$ and $|I_1| + |I_2| \leq t$, and generates random shares for parties with indices in I_2 given shares in S_1 if exists, and $\perp$ otherwise. The algorithm Alg_4 runs a similar process as Alg_1 to find the solution space of $(\boldsymbol{s}, \boldsymbol{r})$, and then computes the subspace $(\boldsymbol{o}, \boldsymbol{r}_0)$ by restricting $\boldsymbol{s}$ to be an all-zero vector. Then the algorithm Alg_4 picks a random $\boldsymbol{r}_0$ and outputs $\pi_{I_2}(\Sigma.\mathsf{Sh}(\boldsymbol{o}, \boldsymbol{r}_0))$. Note that from the t-privacy, if the solution space of $(\boldsymbol{s}, \boldsymbol{r})$ is not empty, then the distribution of shares with indices in I_2 generated from the solution space of $(\boldsymbol{s}, \boldsymbol{r})$ is identical to that generated from the subspace $(\boldsymbol{o}, \boldsymbol{r}_0)$.

3.2 Garbling Scheme

In our construction of constant-round protocols, we follow the definition of garbling schemes and use them in a black-box way. Intuitively, garbling schemes model the algorithms of garbling a circuit and its input, how to evaluate the garbled circuit, and how to decode the circuit output. A formal definition of garbling schemes is given in [BHR12].

Definition 4. (Circuit [BHR12]). *A circuit is a six-tuple $f = (f.n, f.m, q, A, B, G)$. Here $f.n \geq 2$ is the number of inputs, $f.m \geq 1$ is the number of outputs, and $q \geq 1$ is the number of gates. We let $r = f.n + q$ be the number of wires. We let $\mathtt{Inputs} = \{1, \ldots, f.n\}$, $\mathtt{Wires} = \{1, \ldots, f.n + q\}$, $\mathtt{Outputs} = \{f.n+q-f.m+1, \ldots, f.n+q\}$, and $\mathtt{Gates} = f.n + 1, \ldots, f.n + q$. Then $A : \mathtt{Gates} \to \mathtt{Wires} \setminus \mathtt{OutputWires}$ is a function to identify each gate's first incoming wire and $B : \mathtt{Gates} \to \mathtt{Wires} \setminus \mathtt{OutputWires}$ is a function to identify each gate's second incoming wire. Finally, $G : \mathtt{Gates} \times \{0,1\}^2 \to \{0,1\}$ is a function that determines the functionality of each gate. We require $A(g) \leq B(g) < g$ for all $g \in \mathtt{Gates}$.*

Definition 5. (Garbling Scheme [BHR12]). *A garbling scheme is a tuple of five algorithms $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$. The first of these is probabilistic and the remaining algorithms are deterministic. A string f, the original function, describes the function $\mathsf{ev}(f, \cdot) : \{0,1\}^{f.n} \to \{0,1\}^{f.m}$ that we want to garble. On input f and a security parameter κ, algorithm Gb returns a triple of strings $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$. String e describes an encoding function, $\mathsf{En}(e, \cdot)$, that maps an initial input $x \in \{0,1\}^{f.n}$ to a garbled input $X = \mathsf{En}(e, x)$. String F describes a garbled function, $\mathsf{Ev}(F, \cdot)$, that maps each garbled input X to a garbled output $Y = \mathsf{Ev}(F, X)$. String d describes a decoding function, $\mathsf{De}(d, \cdot)$, that maps a garbled output Y to a final output $y = \mathsf{De}(d, Y)$. $\mathcal{G}$ is required to satisfy the following condition:*

- *The non-degeneracy condition: If f, f' describe functions with the same input and output length, i.e. $f.n = f'.n$ and $f.m = f'.m$, and if $|f| = |f'|$, it holds that $\mathsf{Gb}(1^\kappa, f) \equiv \mathsf{Gb}(1^\kappa, f')$.*
- *The correctness condition: For any $f \in \{0,1\}^*, k \in \mathbb{N}, x \in \{0,1\}^{f.n}$, and $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$, $\mathsf{De}(d, \mathsf{Ev}(F, \mathsf{En}(e, x))) = \mathsf{ev}(f, x)$.*

If f is interpreted as a circuit, the garbling scheme is called a circuit garbling scheme.

Additionally, we give the definition of a private garbling scheme. The privacy of a garbling scheme guarantees that the garbled circuit (together with the decoding function) and the garbled inputs won't leak any information about the circuit input.

Definition 6. (Private Garbling Scheme [BHR12], Modified). *A garbling scheme* $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ *is said to be private if there exists a PPT algorithm* Sim *such that, for any circuit* f *with input* $x \in \{0,1\}^{f.n}$*, it holds that*

$$\mathsf{Sim}(1^\kappa, f, y) \equiv_c (F, X, d)$$

for $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$ *and* $\mathsf{En}(e, x) = X$ *with* $\mathsf{ev}(f, x) = y$*.*

In this work, we require the garbling schemes we used to be *projective*, which is the common approach of the existing garbling schemes. In a projective garbling scheme, each input wire is associated with two labels that respectively encode zero and one.

Definition 7. (Projective Scheme [BHR12], Modified). *A garbling scheme* $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ *is said to be projective if for all* f *with* $x = (x_1, \ldots, x_{f.n}) \in \{0,1\}^{f.n}, \kappa \in \mathbb{N}$ *and* $i \in \{1, \ldots, f.n\}$*, whenever* $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$*,* e *is of form* $((X_{1,0}, X_{1,1}), \ldots, (X_{f.n,0}, X_{f.n,1}))$ *with each* $|X_{i,0}| = |X_{i,1}|$ *and* $\mathsf{En}(e, x) = (X_{1,x_1}, \ldots, X_{f.n,x_{f.n}})$*.*

Note that for a projective garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$, the encoding algorithm En can be expressed as $\mathsf{En}_1, \ldots, \mathsf{En}_{f.n}$ such that

$$\mathsf{En}(e, (x_1, \ldots, x_{f.n})) = (\mathsf{En}_1(e, x_1), \ldots, \mathsf{En}_{f.n}(e, x_{f.n})).$$

3.3 Symmetric Key Encryption Scheme

Now we introduce the general notion of a symmetric key encryption scheme.

Definition 8. *A symmetric key encryption scheme consists of three algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *defined below:*

- $k \leftarrow \mathsf{Gen}(1^\kappa)$*. The PPT key generation algorithm* Gen *takes as input a security parameter* κ *and generates a secret key* k*.*
- $c \leftarrow \mathsf{Enc}(k, m)$*. The PPT encryption algorithm* Enc *takes as input a key* k *and a message* m*, and outputs a ciphertext* c*.*
- $m \leftarrow \mathsf{Dec}(k, c)$*. The deterministic polynomial time decryption algorithm* Dec *takes as input a key* k *and a ciphertext* c*, and outputs a plaintext message* m*.*

Additionally, symmetric key encryption must satisfy the following properties:

- **Correctness***: For all security parameters* κ *and all messages* m *it hold that:*

$$\Pr[m = \mathsf{Dec}(k, c) : k \leftarrow \mathsf{Gen}(1^\kappa), c \leftarrow \mathsf{Enc}(k, m)] = 1.$$

- **Security***: For all security parameters* κ *and every choice of vectors* $(m_0^{(1)}, \ldots, m_0^{(q)})$ *and* $(m_1^{(1)}, \ldots, m_1^{(q)})$*, where* $q = \mathsf{poly}(\kappa)$*, it holds that:*

$$\{\{\mathsf{Enc}(k, m_0^{(i)})\}_{i=1}^q : k \leftarrow \mathsf{Gen}(1^\kappa)\} \equiv_c \{\{\mathsf{Enc}(k, m_1^{(i)})\}_{i=1}^q : k \leftarrow \mathsf{Gen}(1^\kappa)\}.$$

3.4 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:

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

4 Constant-Round MPC from Black-Box Garbling

In this section, we provide a compiler from an abstract MPC protocol to a constant-round MPC protocol with a black-box use of garbling schemes.

4.1 Abstract Non-Constant-Round MPC Protocol

We consider an abstract non-constant-round protocol Π_0 that is built upon an (n, t, k, ℓ)-LSSS Σ over $\mathbb{F}_2$ with 3-multiplicative reconstruction. Let the associated LSSSs be $\Sigma^{(2)}, \Sigma^{(3)}$. We use $[\cdot], [\cdot]^{(2)}, [\cdot]^{(3)}$ to denote Σ-sharings, $\Sigma^{(2)}$-sharings, and $\Sigma^{(3)}$-sharings respectively.

We first list the properties we need Π_0. Later in Section F, we give a concrete instantiation of the desired abstract MPC protocol.

- Π_0 runs among m clients and n servers. Only clients have inputs or receive outputs.
- Π_0 is constructed in the $\{\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}\}$-hybrid model. The parties don't have inputs to $\mathcal{F}_{\mathsf{prep}}$ and only input their function inputs to $\mathcal{F}_{\mathsf{input}}$. The functionalities $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ are not restricted to have certain forms of outputs for the abstract protocol. The clients and servers first invoke $\mathcal{F}_{\mathsf{prep}}$, then invoke $\mathcal{F}_{\mathsf{input}}$ with their circuit input, and then run an Evaluation Phase.
- The evaluation phase only involves local computation and reconstructions of $\Sigma^{(2)}$-sharings (by letting each server send his share to the receiver), the receiver runs the algorithm Alg_1 of Remark 1 to reconstruct the secret of each $\Sigma^{(2)}$-sharing after receiving the shares from $n-t$ parties (the secret is unique from Lemma 1). If the algorithm returns $\perp$, the party aborts the protocol. We further require that each honest party's (including both clients and servers) local computation only depends on his output from $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$, and the secrets of $\Sigma^{(2)}$-sharings reconstructed to him in the evaluation phase.
- Suppose that the reconstruction process needs to be done rec times. Let the $\Sigma^{(2)}$-sharings to be reconstructed be $[\boldsymbol{s}_1]^{(2)}, \ldots, [\boldsymbol{s}_{\mathsf{rec}}]^{(2)}$. We assume Π_0 satisfies the following variant of the malicious security:
  - In the real world, an adversary $\mathcal{A}$ may corrupt a subset $\mathcal{C}$ of clients and servers, and let $\mathcal{H}$ be the set of honest clients and servers. The adversary has control of the behaviors of parties in $\mathcal{C}$ during $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ and then fail-stops before the evaluation phase. Since the secret of each $\Sigma^{(2)}$-sharing is uniquely determined by the $n-t$ honest servers' shares by Lemma 1, the evaluation phase can be performed without the t corrupted servers. In particular, we require that all the honest parties (including both honest clients and honest servers) either abort the protocol during the invocation of $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ or never abort. The output in the real-world execution is defined by

$$\mathsf{REAL}_{\Pi_0,\mathcal{A},\mathcal{C}}(\boldsymbol{x}) = (\mathsf{View}_{\mathcal{C}}(\boldsymbol{x}), ([\boldsymbol{s}_1]^{(2)}, \ldots, [\boldsymbol{s}_{\mathsf{rec}}]^{(2)})_{\mathcal{H}}, \mathsf{output}^{\Pi_0}_{\mathcal{H}}(\boldsymbol{x})),$$

where $([\boldsymbol{s}_1]^{(2)}, \ldots, [\boldsymbol{s}_{\mathsf{rec}}]^{(2)})_{\mathcal{H}}$ means the honest servers' shares of $[\boldsymbol{s}_1]^{(2)}, \ldots, [\boldsymbol{s}_{\mathsf{rec}}]^{(2)}$, and $\mathsf{output}_{\mathcal{H}}^{\Pi_0}(\boldsymbol{x})$ means the outputs of honest clients obtained from the execution of Π_0.

– In the ideal world, an ideal adversary Sim_0 controls the same subset $\mathcal{C}$ of clients and servers and has one-time access to the ideal functionality $\mathcal{F}$. The output in the ideal-world execution is defined by the output of Sim_0 and the output of honest clients, denoted by $\mathsf{IDEAL}_{\mathcal{F},\mathsf{Sim}_0,\mathcal{C}}(\boldsymbol{x})$.

We require that for all PPT adversary $\mathcal{A}$ and set $\mathcal{C}$ that may contain any number of clients and at most t servers, there exists a PPT adversary Sim_0 such that

$$\mathsf{REAL}_{\Pi_0,\mathcal{A},\mathcal{C}}(\boldsymbol{x}) \equiv_c \mathsf{IDEAL}_{\mathcal{F},\mathsf{Sim}_0,\mathcal{C}}(\boldsymbol{x}).$$

Remark 2. *The last property deviates from the standard malicious security from the following two aspects.*

First, the adversary can only control the behaviors of corrupted parties during $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ but not the evaluation phase. This means that the adversary is still allowed to change corrupted parties' inputs (in $\mathcal{F}_{\mathsf{input}}$), send malicious instructions to $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$, or receive backdoor information from $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$.

Second, we require that honest parties' shares of all $\Sigma^{(2)}$-sharings in the evaluation phase can be made public. We note that this property can be assumed without loss of generality: For each reconstruction of $[\boldsymbol{s}]^{(2)}$ that should be reconstructed to a receiver R, $\mathcal{F}_{\mathsf{prep}}$ samples and distributes a random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}]^{(2)}$ to all parties and send the whole sharing to R. Then in the evaluation phase, the servers compute and send $[\boldsymbol{s}+\boldsymbol{r}]^{(2)}$ to R for reconstruction. After reconstructing the secret $\boldsymbol{s}+\boldsymbol{r}$, R locally compute $\boldsymbol{s}$ with $\boldsymbol{s}+\boldsymbol{r}$ and $\boldsymbol{r}$. In this way, the reconstructed $\Sigma^{(2)}$-sharing $[\boldsymbol{s}+\boldsymbol{r}]^{(2)}$ can be made public.

4.2 Towards Constant-Round MPC with Malicious Security

In this section, we provide a compiler that compiles an abstract non-constant-round MPC protocol Π_0 to a constant-round protocol Π_1 via

- A *projective* and private garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ with
$$\mathsf{En}(e, (x_1, \ldots, x_\alpha)) = (\mathsf{En}_1(e, x_1), \ldots, \mathsf{En}_\alpha(e, x_\alpha)),$$
- And a symmetric key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ whose key generation algorithm $\mathsf{Gen}(1^\kappa)$ outputs a κ-bit string as the secret key.

Like Π_0, Π_1 runs among m clients and n servers.

Summary. Recall that the online phase of Π_0 contains rec reconstructions, each server S_j's local circuit can be divided into rec circuits $\mathtt{Circ}_1^{S_j}, \ldots, \mathtt{Circ}_{\mathsf{rec}}^{S_j}$. Each circuit is a Boolean circuit containing input gates, AND gates, XOR gates, and output gates. For $i = 1, \ldots, \mathsf{rec}$, each $\mathtt{Circ}_i^{S_j}$ only outputs S_j's share of the $\Sigma^{(2)}$-sharing for the i-th reconstruction. Each server S_j's local computation can be done by computing $\mathtt{Circ}_1^{S_j}, \ldots, \mathtt{Circ}_{\mathsf{rec}}^{S_j}$ successively. Each client C_i's local computation in the evaluation phase of Π_0 can be regarded as a local circuit $\mathtt{Circ}^{C_i}$ that computes C_i's output of Π_0. We give the protocol Π_1 below.

Protocol Π_1

1. **Preprocessing.** The clients and servers invoke $\mathcal{F}_{\mathsf{prep}}$ and obtain the preprocessing data.
2. **Input.** The clients and servers invoke $\mathcal{F}_{\mathsf{input}}$ and obtain the input data.
3. **Garbling Local Circuits.** For $i = 1, \ldots, \mathsf{rec}$, let the receiver of the i-th reconstruction be R_i, and let the sharing to be reconstructed be $[\boldsymbol{s}_i]^{(2)}$. The servers do the following:
   (a) R_i runs $\mathsf{Gen}(1^\kappa)$ $2k$ (recall that k is the number of secrets stored in one Σ-sharing) times to get keys $\boldsymbol{r}_{b,\beta} = (r_{b,\beta}^{(1)}, \ldots, r_{b,\beta}^{(\kappa)})$ for $b = 0, 1$ and $\beta = 1, \ldots, k$. R_i then associate $(\boldsymbol{r}_{0,\beta}, \boldsymbol{r}_{1,\beta})$ with the β-th bit of $\boldsymbol{s}_i$. Let $\boldsymbol{r}_{\boldsymbol{b}}^{(\alpha)} = (r_{b,1}^{(\alpha)}, \ldots, r_{b,k}^{(\alpha)})$ for each $b = 0, 1$ and $\alpha = 1, \ldots, \kappa$.
   (b) For $\alpha = 1, \ldots, \kappa$, R_i randomly samples a $\Sigma^{(3)}$-sharing $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}$ and a Σ-sharing $[\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]$ based on

$\boldsymbol{r}_{\mathbf{0}}^{(\alpha)}, \boldsymbol{r}_{\mathbf{1}}^{(\alpha)}$. Then R_i distributes $[\boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\mathbf{1}}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]$ to all the servers.

(c) Let $\mathtt{Rec}_i^{S_j}$ be the circuit that takes S_j's shares of $[\boldsymbol{s}_i]^{(2)}, \{[\boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\mathbf{1}}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]\}_{\alpha=1}^{\kappa}$ as inputs and outputs S_j's shares of

$$[\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}]^{(3)} := [\boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}_{\mathbf{1}}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0}}^{(\alpha)}], \quad \alpha = 1, \ldots, \kappa$$

and $[\boldsymbol{s}_i]^{(2)}$. Then, let $\mathtt{C}_i^{S_j}$ be the circuit which computes $\mathtt{Circ}_i^{S_j}$ first and then computes $\mathtt{Rec}_i^{S_j}$ (with input wires from preprocessing/input phase or previously reconstructed values, and outputting S_j's shares of $[\boldsymbol{s}_i]^{(2)}$ and $\{[\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}]^{(3)}\}_{\alpha=1}^{\kappa}$).

(d) Each server S_j computes $\mathsf{Gb}(1^\kappa, \mathtt{C}_i^{S_j}) = (\mathtt{GC}_i^{S_j}, e_i^{S_j}, d_i^{S_j})$ and sends $\mathtt{GC}_i^{S_j}, d_i^{S_j}$ to P_{king}.

4. **Encrypting Input Labels.** For each $i = 1, \ldots, \mathsf{rec}$, if the receiver R_i is a server and the β-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_β in R_i's circuit $\mathtt{Circ}_\gamma^{R_i}$ ($\gamma \in \{i+1, \ldots, \mathsf{rec}\}$), R_i computes

$$\mathsf{ct}_{j_\beta,0}^{(i,\gamma)} = \mathsf{Enc}(\boldsymbol{r}_{0,\beta}, \mathsf{En}_{j_\beta}(e_\gamma^{R_i}, 0))$$

and

$$\mathsf{ct}_{j_\beta,1}^{(i,\gamma)} = \mathsf{Enc}(\boldsymbol{r}_{1,\beta}, \mathsf{En}_{j_\beta}(e_\gamma^{R_i}, 1)),$$

where $\boldsymbol{r}_{0,\beta}, \boldsymbol{r}_{1,\beta}$ are the output of Gen that are associated with $\boldsymbol{s}_i$. Then, R_i sends $\{\mathsf{ct}_{j_\beta,0}^{(i,\gamma)}, \mathsf{ct}_{j_\beta,1}^{(i,\gamma)}\}$ to P_{king}.

5. **Sending Input Labels.** Each server S_j computes $\mathsf{En}_\gamma(e_i^{S_j}, x_\gamma)$ for each $i = 1, \ldots, \mathsf{rec}$ and each input wire value x_γ for the γ-th input wire of $\mathtt{C}_i^{S_j}$ where x_γ does not come from reconstructions (i.e. x_γ may come from $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$, or x_γ may be a bit of S_j's share of $[\boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]^{(3)}$ or $[\boldsymbol{r}_{\mathbf{1}}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]$ from a receiver of reconstruction). Then S_j sends them to P_{king}.

6. **Evaluating the Circuit.** The evaluator P_{king} evaluates the circuit from the garbled inputs for wire values that do not come from reconstruction. The concrete evaluation process Ev_0 is given in Figure 2.

7. **Sending Outputs.**

(a) For each client receiver R_i, P_{king} sends $\boldsymbol{s}_i, \{\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}\}_{\alpha=1}^{\kappa}$ to R_i.

(b) Each client receiver R_i checks whether $\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)} = \boldsymbol{r}_{\mathbf{0}}^{(\alpha)} + \boldsymbol{s}_i * (\boldsymbol{r}_{\mathbf{1}}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0}}^{(\alpha)})$ holds for each pair of $(\boldsymbol{r}_{\mathbf{0}}^{(\alpha)}, \boldsymbol{r}_{\mathbf{1}}^{(\alpha)})$ generated in Step 3.(a). If not, R_i aborts the protocol. Otherwise, R_i computes $\mathtt{Circ}^{R_i}$ to get his output locally.

Figure 1: The protocol Π_1.

Process Ev_0

P_{king} does the following to evaluate the circuit:

1. For $j = 1, \ldots, n$, since no input to $\mathtt{C}_1^{S_j}$ comes from reconstruction, P_{king} already gets the garbled input $X_1^{S_j} = \mathsf{En}(e_1^{S_j}, x_1^{S_j})$ for input $x_1^{S_j}$ of circuit $\mathtt{C}_1^{S_j}$. Then, P_{king} obtains S_j's shares of $[\boldsymbol{s}_1]^{(2)}$ and $\{[\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}]^{(3)}\}_{\alpha=1}^{\kappa}$ by computing $\mathsf{De}(d_1^{S_j}, \mathsf{Ev}(\mathtt{GC}_1^{S_j}, X_1^{S_j}))$. After computing all the servers' shares, P_{king} checks whether the sharings $[\boldsymbol{s}_1]^{(2)}, \{[\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}]^{(3)}\}_{\alpha=1}^{\kappa}$ are valid $\Sigma^{(2)}, \Sigma^{(3)}$ sharings respectively. If not, P_{king} aborts the protocol. Otherwise, P_{king} reconstructs $\boldsymbol{s}_1$ and $\{\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}\}_{\alpha=1}^{\kappa}$.
2. The garbled input of each input wire of $\mathtt{C}_2^{S_1}, \ldots, \mathtt{C}_2^{S_n}$ associated with values from $\boldsymbol{s}_1$ can be decrypted by using $\boldsymbol{s}_1, \boldsymbol{r}_{\boldsymbol{s}_1}$ and the corresponding ciphertexts. Thus, P_{king} can evaluate $\mathtt{C}_2^{S_1}, \ldots, \mathtt{C}_2^{S_n}$ in the same way as Step (a). Repeating the above steps, P_{king} eventually obtains all the secrets of the $\Sigma^{(2)}$-sharings $[\boldsymbol{s}_i]^{(2)}$ and their corresponding $\Sigma^{(3)}$-sharings $\{[\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}]^{(3)}\}_{\alpha=1}^{\kappa}$ whose receiver R_i is a client.

Figure 2: The evaluation process for the evaluator P_{king} of Π_0.

Theorem 3. *Let Σ be an (n, t, k, ℓ)-LSSS over $\mathbb{F}_2$ with 3-multiplicative reconstruction and Π_0 be a protocol that has the properties listed in Section 4.1. Protocol Π_1 securely realizes $\mathcal{F}$ in the $\{\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}\}$-hybrid model against a fully malicious adversary that corrupts any number of clients and at most t servers.*

We give the proof of this theorem in Section C.

5 Boosting the Efficiency via Concrete Garbling Schemes

In this section, we analyze the effect of the garbling scheme $\mathcal{G}$ on the communication complexity of the protocol Π_1 and give a concrete instantiation for the garbling scheme $\mathcal{G}$. Based on a concrete garbling scheme, we modify Π_1 to boost its efficiency.

5.1 The Choice of the Garbling Scheme

We first analyze how the choice of the garbling scheme $\mathcal{G}$ affects the communication efficiency. In Π_1,

- Each server S_j needs to send to P_{king} his garbled circuits with decoding data $(\mathtt{GC}_i^{S_j}, d_i^{S_j}), i = 1, \ldots, \mathsf{rec}$ (Step 3(d));
- For each input of S_j's circuits, if this input is from reconstruction, S_j needs to send a pair of ciphertexts to P_{king} (Step 4); Otherwise, S_j needs to send the corresponding garbled input to P_{king} (Step 5).

Therefore, a smaller size of garbled circuits and garbled inputs leads to a smaller communication cost.

The existing projective garbling schemes [BMR90, NPS99, KS08, PSSW09, KMR14, ZRE15] are mostly based on Yao's garbled circuit [Yao86]. The idea of Yao's scheme is to garble the circuit gate by gate by choosing wire labels for each wire of the circuit and encrypting the output wire label of each gate with the input wire labels, and the size of the garbled circuit F grows linearly in the number of gates. To make our protocol Π_1 efficient, we utilize the FreeXOR technique in garbling schemes [KS08]. In this way, the size of the garbled circuit only grows linearly with the number of input and multiplication gates.

The FreeXOR technique enables the garbler not to garble the XOR gates. The idea is to let the XOR $k_{w,0} \oplus k_{w,1} = \Delta$ of the two labels $k_{w,0}, k_{w,1}$ be the same for each wire w. Then, for each XOR gate with input wire a, b and output wire c, by setting $k_{c,0} = k_{a,0} \oplus k_{b,0}$, there is no need to compute any ciphertexts for XOR gates and the evaluator can compute the desired output wire label locally. Using garbling schemes with the FreeXOR property, the communication cost of Π_1 won't be affected by the number of XOR gates in each server's circuits for local computation. For convenience, we just choose the garbling scheme in [KS08] to instantiate our protocol. The more advanced optimizations on the FreeXOR garbling scheme [ZRE15, RR21] can also be applied in our construction.

Remark 3. *All the existing optimizations on the FreeXOR garbling scheme are on the constant factor, the size of the garbled circuit F outputted by $\mathsf{Gb}(1^\kappa, f)$ is still $O((G_I + G_A) \cdot \kappa)$, where G_I is the number of input wires and G_A is the number of AND gates in the boolean circuit f.*

5.2 Boosting the Efficiency

Relying on the properties of the garbled circuits proposed above, we can further improve the concrete efficiency of Π_1 from the following two points.

Reducing the Size of Garbled Circuits. Recall that in Step 2.(c) of Π_1, each server S_j needs to garble the circuit $\mathtt{Circ}_i^{S_j} \circ \mathtt{Rec}_i^{S_j}$ which computes his shares of

$$[\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}]^{(3)} := [\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}], \quad \alpha = 1, \ldots, \kappa$$

and $[\boldsymbol{s}_i]^{(2)}$. Compared with $\mathtt{Circ}_i^{S_j}$, $\mathtt{Rec}_i^{S_j} \circ \mathtt{Circ}_i^{S_j}$ also takes S_j's shares of $\{[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]\}_{\alpha=1}^{\kappa}$ as input and adds $\ell^3 \cdot \kappa$ multiplication gates, where recall that ℓ^3 is the share size of a $\Sigma^{(3)}$-sharing. This in total brings additional $O(\ell^3 \cdot \kappa^2)$ overhead in each garbled circuit.

Our first optimization is to only garble $\mathtt{Circ}_i^{S_j}$ but not $\mathtt{Rec}_i^{S_j}$ so that we can avoid the above overhead. To better explain our idea, we abuse the notation and use $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \boldsymbol{u}$ to denote S_j's shares of

$\{[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}\}_{\alpha=1}^{\kappa}, [\boldsymbol{s}_i]^{(2)}, \{[\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]\}_{\alpha=1}^{\kappa}, \{[\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}]^{(3)}\}_{\alpha=1}^{\kappa}$ respectively. Then $\boldsymbol{x}, \boldsymbol{u} \in \mathbb{F}_2^{\ell^3 \cdot \kappa}$, $\boldsymbol{y} \in \mathbb{F}_2^{\ell^2}$, $\boldsymbol{z} \in \mathbb{F}_2^{\ell \cdot \kappa}$, and $\boldsymbol{u} = \boldsymbol{x} + \boldsymbol{y} \otimes \boldsymbol{z}$. We further split $\boldsymbol{x}, \boldsymbol{u}$ into ℓ^2 sub-vectors, each of ℓ bits, denoted by $\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{\ell^2})$ and $\boldsymbol{u} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{\ell^2})$. Let y_i denote the i-th bit of $\boldsymbol{y}$. Then for all $i \in \{1, \ldots, \ell^2\}$, we have

$$\boldsymbol{u}_i = \boldsymbol{x}_i + y_i \cdot \boldsymbol{z}.$$

Then we may view $\boldsymbol{u}_i$ as the wire label for y_i where $\boldsymbol{u}_i = \boldsymbol{x}_i$ if $y_i = 0$ and $\boldsymbol{u}_i = \boldsymbol{x}_i + \boldsymbol{z}$ otherwise. Note that $\boldsymbol{y}$ is the output of $\mathtt{Circ}_i^{S_j}$. Thus, our idea is to use a garbling scheme where we can choose the labels for the output wires so that at the end of the evaluation, the evaluator can learn not only the output wire values but also the corresponding wire labels. This property can be easily achieved from the garbling scheme [KS08] by adding an output gate to each output wire of the circuit, and then we can use the input wire labels (which are the output wire labels of the original garbled circuit) of each output gate to encrypt the output labels we select.

By using such a garbling scheme with selective output labels, we only need to garble $\mathtt{Circ}_i^{S_j}$ and thus reduce the overhead from $O(\ell^3 \cdot \kappa^2)$ to $O(\ell^3 \cdot \kappa)$. Now we formally present the garbling algorithm and the evaluation algorithm of the garbled circuit as follows. To overcome the issue of circular encryption, we give the garbling scheme under the assumption of random oracles.

Algorithm $\mathsf{Gb}(\cdot, \cdot, \cdot)$

Public Parameter: We assume a random oracle $\mathcal{O}_1$ with output length κ and a random oracle $\mathcal{O}_2$ with output length $\ell\kappa$.

Input: The security parameter 1^κ, a circuit f which describes a function: $\mathsf{ev}(f, \cdot) : \{0,1\}^{f.n} \to \{0,1\}^{f.m}$ to be garbled, and a vector $L = ((Y_{1,0}, Y_{1,1}), \ldots, (Y_{f.m,0}, Y_{f.m,1}))$ of output labels, where each label $Y_{a,b}$ is of length $\ell\kappa$.

The algorithm $\mathsf{Gb}(1^\kappa, f, L)$ runs the following:

1. Sample a random $(\kappa - 1)$-bit string Δ.
2. For each wire w in f that is not an output wire of an XOR gate or an output gate, sample a random masking bit $\lambda_w \in \{0,1\}$ and a random $(\kappa - 1)$-bit string $k_{w,0}$.
3. For each XOR gate in f with input wires a, b and output wire o, compute $k_{o,0} = k_{a,0} \oplus k_{b,0}$ and $\lambda_o = \lambda_a \oplus \lambda_b$ gate by gate from the first layer.
4. For each wire w in f that is not an output wire of an output gate, compute $k_{w,1} = k_{w,0} \oplus \Delta$.
5. For each AND gate g in f with input wire a, b and output wire o and for each $(i, j) \in \{(0,0), (0,1), (1,0), (1,1)\}$, let $\chi = (i \oplus \lambda_a) \wedge (j \oplus \lambda_b) \oplus \lambda_c$. Query the random oracle $\mathcal{O}_1$ with $k_{a,i} \| i \| k_{b,j} \| j \| g$ and then compute the following ciphertext:
$$\mathsf{ct}_{i,j}^{(g)} = \mathcal{O}_1(k_{a,i} \| i \| k_{b,j} \| j \| g) \oplus k_{o,\chi} \| \chi.$$
6. For each output gate in f indexed $k = 1, \ldots, f.m$ with input wire w (we can regard that an output gate has two input wires that are the same to match the definition of a circuit) and for each $i = 0, 1$, query the random oracle $\mathcal{O}_2$ with $k_{w,i} \| i \| w$ and then compute the following ciphertext:
$$\mathsf{ct}_{w,i} = \mathcal{O}_2(k_{w,i} \| i \| w) \oplus Y_{k, i \oplus \lambda_w}.$$
7. Let the F be the set of all the ciphertexts. Let $X_{i,0} = k_{w_i, \lambda_{w_i}} \| \lambda_{w_i}, X_{i,1} = k_{w_i, 1 \oplus \lambda_{w_i}} \| (1 \oplus \lambda_{w_i})$ for each $i = 1, \ldots, f.n$ and $e = (X_{1,0}, X_{1,1}, \ldots, X_{n,0}, X_{n,1})$, where w_i is the i-th input wire of f. Let d be the set of λ_w for each input wire w of output gates in f.
8. Output (F, e, d).

Figure 3: The garbling algorithm.

Algorithm $\mathsf{Ev}(\cdot,\cdot,\cdot)$

Input: A garbled circuit F, a garbled input X with $X = \mathsf{En}(e, x)$, and d for $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f, L)$[1].

The algorithm $\mathsf{Ev}(F, X, d)$ runs the following:

1. Evaluate the garbled circuit gate by gate from the first layer. While evaluating the gates in one layer, the input wire labels $k_{w,v_w\oplus\lambda_w}\|v_w \oplus \lambda_w$ for each input wire w with value v_w of these gates have been computed. More concretely, for each gate g in the circuit:
   - If g is an XOR gate with input wire a, b and output wire o, compute $k_{o,v_o\oplus\lambda_o}\|v_o \oplus \lambda_o = (k_{a,v_a\oplus\lambda_a}\|(v_a \oplus \lambda_a)) \oplus (k_{b,v_b\oplus\lambda_b}\|(v_b \oplus \lambda_b))$.
   - If g is an AND gate with input wire a, b and output wire o, query the random oracle $\mathcal{O}_1$ with $k_{a,v_a\oplus\lambda_a}\|(v_a \oplus \lambda_a)\|k_{b,v_b\oplus\lambda_b}\|(v_b \oplus \lambda_b)\|g$ and then compute
   $$\begin{aligned} &k_{o,v_o\oplus\lambda_o}\|v_o \oplus \lambda_o \\ =&\mathcal{O}_1(k_{a,v_a\oplus\lambda_a}\|(v_a \oplus \lambda_a)\|k_{b,v_b\oplus\lambda_b}\|(v_b \oplus \lambda_b)\|g) \oplus \mathsf{ct}^{(g)}_{v_a\oplus\lambda_a, v_b\oplus\lambda_b}. \end{aligned}$$
   - If g is an output gate with input wire w, query the random oracle $\mathcal{O}_2$ with $k_{w,v_w\oplus\lambda_w}\|(v_w \oplus \lambda_w)\|w$ and compute
   $$Y_{k,v_w} = \mathcal{O}_2(k_{w,v_w\oplus\lambda_w}\|(v_w \oplus \lambda_w)\|w) \oplus \mathsf{ct}_{w,v_w\oplus\lambda_w a}.$$
2. For each input wire w of output gates in f, compute $v_w = (v_w \oplus \lambda_w) \oplus \lambda_w$. Let y be the set of v_w and let Y be the set of Y_{k,v_w} for all output wires w of all output gates (with indices $k = 1, \ldots, f.m$).
3. Output (y, Y).

[1]Here the input F for Ev may also be a garbled sub-circuit of f, where the input x may also be a partial input to f which determines the output of the sub-circuit.

Figure 4: The evaluation algorithm.

Reusing Wire Labels for the Same Wires. With the first optimization, each server S_j only needs to garble $\{\mathtt{Circ}_i^{S_j}\}_{i=1}^{\mathsf{rec}}$, where rec is the total number of reconstructions in the abstract protocol Π_0. By assumption, the inputs of each $\mathtt{Circ}_i^{S_j}$ are from S_j's output in the preprocessing phase and the input phase, and the secrets of $\Sigma^{(2)}$-sharings reconstructed to him in the evaluation phase of Π_0.

We note that the same wire may appear in different circuits while their wire labels may be different when we perform the garbling algorithm on different circuits. For example, assuming that a single wire w serves as the i_1, i_2-th input wires for $\mathtt{Circ}_1^{S_j}, \mathtt{Circ}_2^{S_j}$, the encoding functions $\mathsf{En}_{i_1}(e_1^{S_j}, \cdot), \mathsf{En}_{i_2}(e_2^{S_j}, \cdot)$ for this wire value may be different. However, for most garbling schemes including the one from [KS08], the labels of a single wire can be reused in evaluating different circuits. That is to say, we can view a server S_j's local circuits $\mathtt{Circ}_1^{S_j}, \ldots, \mathtt{Circ}_{\mathsf{rec}}^{S_j}$ as an entire circuit $\mathtt{Circ}^{S_j}$. This circuit may not receive all the input wire values at the beginning. Instead, it can evaluate a gate in it as long as the input wire values of this gate are known. Besides, even if a partial input has not been provided to the circuit, it can generate the output for an output wire with those input values that affect the computation of this output provided. In this way, the wires of all the circuits are indexed together and a single wire only has one label. Although the garbled circuit outputted by the garbling algorithm can also be regarded as rec different garbled circuits, the sum of their size only depends on the total number of distinct wires in $\mathtt{Circ}_1^{S_j}, \ldots, \mathtt{Circ}_{\mathsf{rec}}^{S_j}$.

Summary. The modified protocol Π_1' is constructed via the garbling algorithm Gb shown in Figure 3 and the evaluation algorithm Ev shown in Figure 4.

See Figure 5 for the construction.

Protocol Π_1'

1. **Preprocessing.** The clients and servers invoke $\mathcal{F}_{\mathsf{prep}}$ and obtain the preprocessing data.

2. **Input.** The clients and servers invoke $\mathcal{F}_{\mathsf{input}}$ and obtain the input data.
3. **Generating Output Labels.** For $i = 1, \ldots, \mathsf{rec}$, let the receiver of the i-th reconstruction be R_i, and let the sharing to be reconstructed be $[\boldsymbol{s}_i]^{(2)}$. The servers do the following:
   (a) R_i samples $2k$ (recall that k is the number of secrets stored in one Σ-sharing) random κ-bit string as $\boldsymbol{r}_{b,\beta} = (r_{b,\beta}^{(1)}, \ldots, r_{b,\beta}^{(\kappa)})$ for $b = 0, 1$ and $\beta = 1, \ldots, k$. R_i then associate $(\boldsymbol{r}_{0,\beta}, \boldsymbol{r}_{1,\beta})$ with the β-th bit of $\boldsymbol{s}_i$. Let $\boldsymbol{r}_{\boldsymbol{b}}^{(\alpha)} = (r_{b,1}^{(\alpha)}, \ldots, r_{b,k}^{(\alpha)})$ for each $b = 0, 1$ and $\alpha = 1, \ldots, \kappa$.
   (b) For $\alpha = 1, \ldots, \kappa$, R_i randomly samples a $\Sigma^{(3)}$-sharing $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}$ and a Σ-sharing $[\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]$ based on $\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}, \boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)}$. Then R_i distributes $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]$ to all the servers.
   (c) For each server S_j, S_j computes

$$Y_{(i-1)\ell^2+a,0}^{S_j} = \big(([\boldsymbol{r}_{\boldsymbol{0}}^{(1)}]^{(3)})_{[a\ell+1,(a+1)\ell]}^{S_j}, \ldots, ([\boldsymbol{r}_{\boldsymbol{0}}^{(\kappa)}]^{(3)})_{[a\ell+1,(a+1)\ell]}^{S_j}\big)$$

   and

$$\begin{aligned} Y_{(i-1)\ell^2+a,1}^{S_j} =& \big(([\boldsymbol{r}_{\boldsymbol{0}}^{(1)}]^{(3)})_{[a\ell+1,(a+1)\ell]}^{S_j} + ([\boldsymbol{r}_{\boldsymbol{1}}^{(1)} - \boldsymbol{r}_{\boldsymbol{0}}^{(1)}])^{S_j}, \\ & \ldots, ([\boldsymbol{r}_{\boldsymbol{0}}^{(\kappa)}]^{(3)})_{[a\ell+1,(a+1)\ell]}^{S_j} + ([\boldsymbol{r}_{\boldsymbol{1}}^{(\kappa)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\kappa)}])^{S_j}\big), \end{aligned}$$

   where $([\boldsymbol{s}])^{S_j}$ denotes S_j's share of $[\boldsymbol{s}]$ and $([\boldsymbol{s}]^{(3)})_{[c_1,c_2]}^{S_j}$ denotes the vector of the $c_1, c_1+1, \ldots, c_2$-th bits of S_j's share of $[\boldsymbol{s}]^{(3)}$.
4. **Garbling Local Circuits.** For each server S_j, we view the local circuits $\mathtt{Circ}_1^{S_j}, \ldots, \mathtt{Circ}_{\mathsf{rec}}^{S_j}$ as an entire circuit $\mathtt{Circ}^{S_j}$. Each server S_j computes

$$\mathsf{Gb}(1^\kappa, \mathtt{Circ}^{S_j}, ((Y_{1,0}^{S_j}, Y_{1,1}^{S_j}), \ldots, (Y_{\ell^2\mathsf{rec},0}^{S_j}, Y_{\ell^2\mathsf{rec},1}^{S_j}))) = (\mathtt{GC}^{S_j}, e^{S_j}, d^{S_j})$$

   and sends $\mathtt{GC}^{S_j}, d^{S_j}$ to P_{king}. P_{king} obtains $(\mathtt{GC}_1^{S_j}, \ldots, \mathtt{GC}_{\mathsf{rec}}^{S_j})$ from $\mathtt{GC}^{S_j}$, where each $\mathtt{GC}_i^{S_j}$ is the garbled circuit for $\mathtt{Circ}_i^{S_j}$.
5. **Encrypting Input Labels.** For $i = 1, \ldots, \mathsf{rec}$, if R_i is a server and the β-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_β in R_i's circuit $\mathtt{Circ}^{R_i}$, R_i computes

$$\mathsf{ct}_{j_\beta,0}^{(i)} = \mathcal{O}_1(\boldsymbol{r}_{0,\beta} \| 0 \| i \| \beta \| j_\beta) \oplus \mathsf{En}_{j_\beta}(e^{R_i}, 0)$$

   and

$$\mathsf{ct}_{j_\beta,1}^{(i)} = \mathcal{O}_1(\boldsymbol{r}_{1,\beta} \| 1 \| i \| \beta \| j_\beta) \oplus \mathsf{En}_{j_\beta}(e^{R_i}, 1).$$

   where $\boldsymbol{r}_{0,\beta}, \boldsymbol{r}_{1,\beta}$ are the output of Gen that are associated with $\boldsymbol{s}_i$. Then, R_i sends $\{\mathsf{ct}_{\beta,0}, \mathsf{ct}_{\beta,1}\}$ to P_{king}.
6. **Sending Input Labels.** Each server S_j computes $\mathsf{En}_\gamma(e^{S_j}, x_\gamma)$ for each input wire value x_γ for the input wire indexed γ of S_j's local circuits where x_γ comes from $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$. Then S_j sends them to P_{king}.
7. **Evaluating the Circuit.** The evaluator P_{king} evaluates the circuit by doing the following:
   (a) For $j = 1, \ldots, n$, since no input to $\mathtt{Circ}_1^{S_j}$ comes from reconstruction, P_{king} already gets the garbled input $X_1^{S_j}$ for input $x_1^{S_j}$ of circuit $\mathtt{Circ}_1^{S_j}$. Then, P_{king} runs $\mathsf{Ev}(\mathtt{GC}_1^{S_j}, X_1^{S_j}, d^{S_j})$ to obtain S_j's share of $[\boldsymbol{s}_1]^{(2)}$ and $\{Y_{a,s_{1,a}^{S_j}}^{S_j}\}_{a=1}^{\ell^2}$, where $s_{1,a}^{S_j}$ is the a-th bit of S_j's share of $[\boldsymbol{s}_1]^{(2)}$. Then P_{king} reconstructs S_j's shares of $\{[\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}]^{(3)}\}_{\alpha=1}^{\kappa}$ with $\{Y_{a,s_{i,a}^{S_j}}^{S_j}\}_{a=1}^{\ell^2}$. After computing all the servers' shares, P_{king} checks whether the sharings $[\boldsymbol{s}_1]^{(2)}, \{[\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}]^{(3)}\}_{\alpha=1}^{\kappa}$ are valid $\Sigma^{(2)}, \Sigma^{(3)}$ sharings respectively. If not, P_{king} aborts the protocol. Otherwise, P_{king} reconstructs $\boldsymbol{s}_1$ and $\{\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}\}_{\alpha=1}^{\kappa}$.
   (b) The garbled input of each input wire of $\mathtt{Circ}_2^{S_1}, \ldots, \mathtt{Circ}_2^{S_n}$ associated with values from $\boldsymbol{s}_1$ can be decrypted by using $\boldsymbol{s}_1, \{\boldsymbol{r}_{\boldsymbol{s}_1}\}_{\alpha=1}^{\kappa}$ and the corresponding ciphertexts. Thus, P_{king} can evaluate $\mathtt{Circ}_2^{S_1}, \ldots, \mathtt{Circ}_2^{S_n}$ in the same way as Step (a). Repeating the above steps, P_{king} eventually obtains all the secrets of the $\Sigma^{(2)}$-sharings $[\boldsymbol{s}_i]^{(2)}$ and their corresponding $\Sigma^{(3)}$-sharings $\{[\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}]^{(3)}\}_{\alpha=1}^{\kappa}$ whose receiver R_i is a client.

8. **Sending Outputs.**
   (a) For each client receiver R_i, P_{king} sends $\boldsymbol{s}_i, \{\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}\}_{\alpha=1}^{\kappa}$ to R_i.
   (b) Each client receiver R_i checks whether each $\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}$ matches $\boldsymbol{s}_i$ and $(\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}, \boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)})$ generated in Step 3.(a). If not, R_i aborts the protocol. Otherwise, R_i computes $\mathtt{Circ}^{R_i}$ to get his output locally.

Figure 5: The modified protocol Π_1'.

Theorem 4. *Let Σ be an (n, t, k, ℓ)-LSSS over $\mathbb{F}_2$ with 3-multiplicative reconstruction and Π_0 be a protocol that has the properties listed in Section 4.1. Protocol Π_1' securely realizes $\mathcal{F}$ in the $\{\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}\}$-hybrid model against a fully malicious adversary that corrupts any number of clients and at most t servers.*

We give the proof of this theorem in Section D.

Apart from the communication cost of realizing $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$, the communication cost of Π_1' is $O((\mathsf{DS}+G_A)\cdot\kappa + \ell\kappa\cdot\mathsf{CC}_{\mathsf{eval}}^{\Pi_0})$, where DS is the output size of $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$, G_A is the number of AND gates in all the servers' local circuits, and $\mathsf{CC}_{\mathsf{eval}}^{\Pi_0}$ is the communication cost of the evaluation phase of Π_0. We provide a detailed cost analysis in Section E.

6 Instantiation of the Abstract Protocol for SIMD Circuits

To instantiate the abstract protocol, we follow [GPS21, GPS22] which provides an unconditionally secure MPC protocol with constant communication in the number of servers based on packed Shamir secret sharing. Our idea is to replace the packed Shamir secret sharing with Σ-sharing. Like the packed Shamir secret sharing scheme used in their approach, each secret in Σ is also a vector over a finite field $\mathbb{F} = \mathbb{F}_2$ with length $k = \Theta(n)$, where recall that n is the number of servers. At a high level, the idea is to batch the gates of the same type in each layer into groups of k, and each group of gates will be evaluated at the same time. Then the task of evaluating a general circuit is reduced to the following two steps:

- For each group of addition/multiplication gates, given the two input sharings $[\boldsymbol{x}], [\boldsymbol{y}]$, compute the output sharing $[\boldsymbol{z}]$, where $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$ for addition gates and $\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{y}$ for multiplication gates.
- Given output sharings from all previous layers, prepare the input sharings for the current layer.

For simplicity, we focus on computing SIMD (Single Instruction Multiple Data) circuits in this section and refer the readers to Section F for the treatment of a general circuit. We assume the SIMD circuit repeats a single circuit a multiple of k^2 times. Since a group of output wires from a former layer of such a SIMD circuit naturally serves as a group of input wires to a group of gates in a latter layer, we only need to focus on evaluating groups of addition and multiplication gates.

Overview. Since Σ is $\mathbb{F}_2$-linear, addition gates can be evaluated via local computation: Each party simply adds up his two local shares. For multiplication gates, we first utilize the multiplicative property of Σ to compute $[\boldsymbol{z}]^{(2)} = [\boldsymbol{x}] * [\boldsymbol{y}]$. Now it is sufficient to transform $[\boldsymbol{z}]^{(2)}$ to $[\boldsymbol{z}]$ which is similar to the degree-reduction step when using packed Shamir sharings. However, we cannot use the DN-style approach. This is because the DN-style approach requires all servers to send a $\Sigma^{(2)}$-sharing to a king to reconstruct and reshare the secrets while our abstract protocol only allows reconstruction of $\Sigma^{(2)}$-sharings.

To solve this problem, our idea is to transform a batch of k $\Sigma^{(2)}$-sharings each time. Suppose the input sharings are denoted by $[\boldsymbol{z}_1]^{(2)}, \ldots, [\boldsymbol{z}_k]^{(2)}$. We may view the secrets as a matrix $\boldsymbol{Z}$ of size $k \times k$ and the i-th row of $\boldsymbol{Z}$ is $\boldsymbol{Z}_i = \boldsymbol{z}_i$. We use $\boldsymbol{Z}_{\star,j}$ to denote the j-th column of $\boldsymbol{Z}$. Our idea is to first let all servers obtain $[\boldsymbol{Z}_{\star,1}], \ldots, [\boldsymbol{Z}_{\star,n}]$ only using reconstruction of $\Sigma^{(2)}$-sharings. The main observation is that the sharing algorithm of Σ is $\mathbb{F}_2$-linear. Thus, the high-level idea is to view $\Sigma.\mathsf{Sh}$ as a linear circuit and securely compute this circuit over the input $[\boldsymbol{z}_1]^{(2)}, \ldots, [\boldsymbol{z}_k]^{(2)}$. In this way, all servers can obtain $\Sigma^{(2)}$-sharing of the shares of each party. Then all servers reconstruct the shares to each party. To be more concrete,

1. All servers first prepare random $\Sigma^{(2)}$-sharings $[\boldsymbol{r}_1]^{(2)}, \ldots, [\boldsymbol{r}_{n\ell}]^{(2)}$. We will compute $\Sigma.\mathsf{Sh}(\boldsymbol{Z}_{\star,j}, (r_{1,j}, \ldots, r_{n\ell,j}))$ for all j.
2. All servers view $\Sigma.\mathsf{Sh}$ as a linear circuit. Let $\boldsymbol{y}_{\star,j} = \Sigma.\mathsf{Sh}(\boldsymbol{Z}_{\star,j}, (r_{1,j}, \ldots, r_{n\ell,j}))$ which is of size $n\ell$. Then all servers can locally compute $[\boldsymbol{y}_i]^{(2)}$, where $\boldsymbol{y}_i = (y_{i,1}, \ldots, y_{i,k})$. All parties reconstruct $\boldsymbol{y}_i$ to the $\lceil i/\ell \rceil$-th party. As a result, all parties obtain $[\boldsymbol{Z}_{\star,1}], \ldots, [\boldsymbol{Z}_{\star,k}]$.

The above procedure is modeled as $\Pi_{\mathsf{Transpose}}$. To obtain $[\boldsymbol{z}_1], \ldots, [\boldsymbol{z}_k]$, we simply apply $\Pi_{\mathsf{Transpose}}$ on $[\boldsymbol{Z}_{\star,1}], \ldots, [\boldsymbol{Z}_{\star,k}]$.

The Transpose Protocol. We present the formal description of $\Pi_{\mathsf{Transpose}}$. $\Pi_{\mathsf{Transpose}}$ 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 $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$.

Recall that for an LSSS, the sharing algorithm Sh is linear on each field element of its input, so $\Sigma.\mathsf{Sh}_1, \ldots, \Sigma.\mathsf{Sh}_n : \mathbb{F}_2^{k+n\ell} \to \mathbb{F}_2^{\ell}$ are all $\mathbb{F}_2$-linear functions. Suppose that for each $i = 1, \ldots, n$,

$$\begin{aligned}&\Sigma.\mathsf{Sh}_i((s_1, \ldots, s_k), (a_1, \ldots, a_{n\ell}))\\ =&\left(\sum_{j=1}^{k} c_{1,j}^{(i)} s_j + \sum_{j=1}^{n\ell} c_{1,k+j}^{(i)} a_j, \ldots, \sum_{j=1}^{k} c_{\ell,j}^{(i)} s_j + \sum_{j=1}^{n\ell} c_{\ell,k+j}^{(i)} a_j\right).\end{aligned}$$

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

$$F_i(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{k+n\ell}) = \left(\sum_{j=1}^{k+n\ell} c_{1,j}^{(i)} \boldsymbol{v}_j, \ldots, \sum_{j=1}^{k+n\ell} c_{\ell,j}^{(i)} \boldsymbol{v}_j\right)$$

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 $[\boldsymbol{x}_1]^{(2)}, \ldots, [\boldsymbol{x}_k]^{(2)}$. 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 take a group of $\Sigma^{(2)}$ sharings $[\boldsymbol{r}_1]^{(2)}, \ldots, [\boldsymbol{r}_n]^{(2)}$ (where each server S_j holds the secret $\boldsymbol{r}_j$) and $n\ell$ random $\Sigma^{(2)}$-sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ from the output of $\mathcal{F}_{\mathsf{prep}}$ and associate them with the execution of $\Pi_{\mathsf{Transpose}}$.
2. 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 $i = 1, \ldots, n$.
3. For each $i = 1, \ldots, n$, each server sends his share of $[\boldsymbol{y}_i]^{(2)}$ to server S_i for reconstruction.
4. Each server S_j reconstructs $\boldsymbol{y}_j$ using the algorithm Alg_1 of Remark 1 with the first $n - t$ received shares of $[\boldsymbol{y}_i]^{(2)}$ and computes the vector of his shares of $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$ by $\boldsymbol{y}_i - \boldsymbol{r}_i$.

Figure 6: Protocol to turn row $\Sigma^{(2)}$-sharings to column Σ-sharings.

Evaluating Multiplication Gates. Now, we show how to utilize $\Pi_{\mathsf{Transpose}}$ to compute groups of multiplication gates. Each k groups of k multiplication gates (k^2 multiplication gates in total) are computed together. This can be done with $O(n^2)$-bit communication by the following protocol Π_{Multi}.

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 $[\boldsymbol{z}_j^*]^{(2)} = [\boldsymbol{1}] \otimes [\boldsymbol{z}_j^*]$ for each $j = 1, \ldots, k$, where $[\boldsymbol{1}]$ 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 7: Protocol to compute batched multiplication gates.

Outline of the Protocol for SIMD Circuits. Now we give an outline of the protocol for SIMD circuits.

1. First, the parties invoke $\mathcal{F}_{\mathsf{prep}}$ to get the preprocessing data, including:
   - Preprocessing for the transpose protocol, i.e. each server S_j generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}_j]^{(2)}$, and all the servers jointly prepare random $\Sigma^{(2)}$-sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{N\ell}]^{(2)}$.
   - The mask sharing for output sharings, i.e. each client C_i generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}]^{(2)}$ for each batch of output wires attached to him. This is used to mask the sharing for the group of output wire values. After being masked, the sharings for output wires can be made public to satisfy the requirements of the abstract protocol listed in Section 4.1.
2. The clients run $\mathcal{F}_{\mathsf{input}}$ to generate input Σ-sharings for each group of input wires.
3. The parties evaluate the circuit gate by gate:
   - **Addition Gates:** For each batch of k addition gates with input sharings $[\boldsymbol{x}], [\boldsymbol{y}]$, all the servers locally compute $[\boldsymbol{x} + \boldsymbol{y}] = [\boldsymbol{x}] + [\boldsymbol{y}]$.
   - **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 $[\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_k]$.
4. After evaluating all the layers of the circuit, the servers get the output sharing $[\boldsymbol{y}]$ for each group of output wires attached to each client C_i. The servers locally compute $[\boldsymbol{y} + \boldsymbol{r}]^{(2)} = [\boldsymbol{1}] \otimes [\boldsymbol{y}] + [\boldsymbol{r}]^{(2)}$ with the corresponding $[\boldsymbol{r}]^{(2)}$ and send it to C_i (where $[\boldsymbol{1}]$ is a public Σ-sharing with an all-1 secret vector). Then C_i reconstructs $\boldsymbol{y} + \boldsymbol{r}$ and computes $\boldsymbol{y} = \boldsymbol{y} + \boldsymbol{r} - \boldsymbol{r}$ to get his output.

We give more details about Π_0 for general circuits in Section F. Here we just state the parameters Π_0 achieves. For the protocol Π_0 (Figure 15) that satisfies all the requirements in Section 4.1, we have $\mathsf{DS} = G_A = \mathsf{CC}_{\mathsf{eval}}^{\Pi_0} = O(|C| + Dn^2 + mn)$, where m is the number of clients. Then the communication cost of Π_1' with Π_0 instantiated will be $\mathsf{CC}^{\Pi_1'} = \mathsf{CC}_{\mathsf{prep}} + \mathsf{CC}_{\mathsf{input}} + O((|C| + Dn^2 + mn) \cdot \kappa)$ without adding the cost of realizing $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$, where D is the circuit depth of C.

In Section F.3.1, we give the details of functionalities, and we give the instantiation of the functionalities in Section H. The achieved total communication cost of realizing $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ is $O(|C| + Dn^2 + mn + n^2\kappa)$. Thus, the total communication of Π_1' is $O((|C| + Dn^2 + mn) \cdot \kappa)$. In addition, we give a detailed analysis of rounds for Π_1' in the plain model in Section I. Let $m = n$ and let each party serve as a client and a server at the same time, we give the following theorem in the standard MPC model.

Theorem 1. *Assuming random oracles, there exists a computationally secure 5-round MPC protocol against a fully malicious adversary controlling up to $n/4$ parties with communication of $O(|C|\kappa + Dn^2\kappa)$, where $|C|$ is the circuit size, D is the circuit depth, and κ is the computational security parameter.*

7 Overview of the Dishonest Majority Constant-Round MPC

Supporting Dishonest Majority via Virtual Servers. To support the dishonest majority setting, we use the party-virtualization technique [Bra87] to take c random servers as a group to emulate a virtual server in Π_1'. Let N be the number of virtual parties that will be determined below. Let Π_0, Π_1' be N-server protocols we constructed in Section F and Section 5 which can tolerate up to $N/4$ corrupted servers. A virtual server is regarded as an honest virtual server if it contains at least one honest real server, and vice versa. We will determine N and c to ensure that at most $N/4$ virtual servers are corrupted.

For a fixed set of $t_s = (1-\epsilon)n$ corrupted servers, the probability that at least one honest server is picked is $1-(1-\epsilon)^c > 1-\exp(-c\epsilon)$. Let $1-(1-\epsilon)^c = c_0$, we only need $c = \Theta(1/\epsilon)$ to be a constant to make c_0 larger than any constant smaller than 1. We take N groups of this form, and we let $X_i = 1$ if all the servers in the i-th group are corrupted, and $X_i = 0$ otherwise. Since these N groups of servers are chosen independently, $X_1, \ldots, X_N$ are independent random variables. For each X_i, the probability that $X_i = 0$ is c_0. Let $X = X_1 + \cdots + X_N$, we have $\mu = \mathbb{E}(X) = (1-c_0)N$. By Chernoff bound, it holds that:

$$P(X \geq 2(1-c_0)N) = P(X \geq 2\mu) \leq e^{-\frac{\mu}{3}} = e^{-\frac{(1-c_0)N}{3}} = e^{-\Omega(N)}.$$

Considering that the adversary may choose the set of corrupted parties after the groups are determined, we take the union bound of all $\binom{n}{t} < 2^n$ possible choices, then the probability that there are more than $2(1-c_0)N$ groups of the N groups containing no honest server is no more than $2^n \cdot e^{-\Omega(N)}$. Taking $7/8 < c_0 < 1$ and $N = \Theta(n+\kappa)$ (such that $(1-\epsilon)^c N \geq n+\kappa$), with overwhelming probability, at least $3/4$ groups of servers contains an honest server.

Then, each group of c servers will emulate a virtual server V_i to participate in Π'_1, and they jointly compute the local computation of V_i and communicate with other virtual servers (where P_{king} is still acted by a single real-world server, which is not required to be honest).

Now we explain how the virtual servers can perform Π'_1. To ensure that the data of each honest virtual server won't be revealed to the adversary, we let the data of each virtual server be additively shared among all the servers emulating it. More concretely, for the preprocessing and input data generated in Π'_1, the parties run preprocessing and input protocols with each message to a server S_j in Π'_1 additively shared among the servers emulating the virtual server V_j. In addition, the receivers of $\Sigma^{(2)}$-sharings also generate the sharings for output labels in such an additively shared way. To perform the local computation of V_i, the servers emulating V_i can simply utilize a dishonest majority MPC protocol such as the SPDZ protocol [DPSZ12] to do the computation and compute an additive sharing (denoted by $\langle\cdot\rangle$) for each value in the internal state of V_i. To send a message a to another virtual server V_j, the servers that act as V_i only need to additively share their shares of $\langle a\rangle$ and distribute them to the servers that act as V_j. Then, the servers that act as V_j can locally compute an additive sharing $\langle a\rangle$. In this way, we can transform Π'_1 to support the dishonest majority.

Multiparty Garbling. Now the remaining problem is the efficiency. Simply applying a dishonest majority MPC protocol on each virtual server's local computation may be very inefficient (or even not work since we assume a random oracle in Π'_1), especially for the computation of Gb and Enc. To let the real-world servers jointly garble a circuit, we utilize the multiparty garbling technique. Since the number of parties in each group is constant, we do not need the multiparty garbling scheme to be scalable. So we simply generalize the multiparty garbling technique in the BMR framework [BMR90, DI05] to support freeXOR under the assumption of random oracles.

Note that the multiparty garbling process requires secure multiplications of additive sharings. Since the additive sharings are distributed among a constant number of parties, we do not require a multiplication protocol with linear communication which requires stronger assumptions such as somewhat homomorphic encryptions [DKL+13] or LPN [RS22]. We follow [GMW87] to let the servers invoke random OTs (ROTs) first and use the results to compute pair-wise products of their shares and add them up.

Towards Malicious Security. To achieve malicious security, a direct way is to apply a maliciously secure MPC to do the multiparty garbling. However, these protocols either require non-symmetric cryptographic assumptions such as linearly homomorphic encryption [BDOZ11] and LPN [RS22] or lead to a non-constant multiplication overhead on the communication complexity [LOS14, KOS16]. Our idea is to use a modified version of the "Watchlist" technique based on MPC-in-the-head [IKOS07, IPS08, FR23] to do the check on the local computation of virtual servers.

More concretely, we choose the committee size c of each virtual server to be the smallest integer with $c_0 = 1-(1-\epsilon)^c > 31/32$. In this way, with overwhelming probability, no more than $2(1-c_0)N < N/16$

virtual servers are completely emulated by corrupted servers. Before doing the local computation of each virtual server, the servers commit their inputs and local randomness. Then, the computation of each virtual server is deterministic given these committed inputs. After the local computation of the virtual servers is completed, each server S checks a random set of $N/16n$ virtual servers. This is done by asking all servers to open their commitments of inputs and randomness for the local computation of the chosen virtual servers and send all the transcripts while emulating the virtual servers to S for verification. Since the computation is deterministic on the committed inputs, S can check whether the computation of these chosen virtual servers is correctly performed.

In this way, if the corrupted parties do not follow the protocol to do the multiparty garbling for over $N/16$ virtual servers that are emulated by at least one honest server, the protocol will proceed only when these virtual servers are not chosen by the ϵn honest servers. The probability that each honest server does not choose these virtual servers is

$$\frac{\frac{15N}{16}}{N} \cdot \frac{\frac{15N}{16}-1}{N-1} \cdot \cdots \cdot \frac{\frac{15N}{16}-\frac{N}{16n}+1}{N-\frac{N}{16n}+1} < \left(1-\frac{1}{16}\right)^{\frac{N}{16n}}.$$

Thus, the probability that the $N/16$ virtual servers that are not chosen by all the honest servers is $(15/16)^{(\epsilon N/16)}$. Recall that $N = \Theta(n+\kappa)$, the probability is negligible.

However, one issue we omitted so far is that corrupted servers may not commit their inputs correctly. In our construction, the input of each virtual party either comes from a real server/client or is generated by the functionality of ROT. For the former case, when S verifies the computation of a virtual party V_i, S also receives the input of V_i from the real server/client to cross-check the correctness of the input. For the latter case, however, an adversary may launch the 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, S 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 $N/16$ virtual servers' ROTs, the /probability of catching the adversary when checking a single virtual server is at least $(1/16) \cdot (1/2)$. Thus, the probability that the attack is caught by any honest server (which checks $\epsilon N/16$ virtual servers in total) is bounded by $(31/32)^{(\epsilon N/16)}$, which is negligible.

In this way, we can ensure that with overwhelming probability at least $13/16$ fraction of virtual servers honestly follow the protocol to do the computation. Among them, the $(1-\epsilon)n$ corrupted server may view the input and computation of $(1-\epsilon)N/16 < N/16$ virtual servers. Thus, there are no less than $3/4$ virtual servers that follow the protocol to perform computation, and the honest servers' inputs of these virtual servers are private to the corrupted servers. Then, we can view that these virtual servers are the honest servers in Π_1', which guarantees the security.

Protocol Outline. Finally, we give an outline of the protocol:

1. First, the parties emulate $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ to generate preprocessing and input data to the virtual servers. Each data to V_j is shared among the servers $S_{j,1}, \ldots, S_{j,c}$ emulating V_j.

2. The servers prepare for the garbling of the virtual servers' local circuits:

   - Each pair of servers $(S_{j,\alpha}, S_{j,\beta})$ emulating the same virtual server V_j prepares random OT instances that are needed in performing multiplication of the additively shared data of V_j.
   - The receivers of $\Sigma^{(2)}$-sharings generate sharings for the chosen output labels. Each label $\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}$ for a virtual server receiver V_j is jointly chosen by all the servers emulating it, i.e. $\boldsymbol{r}_{\boldsymbol{b}}^{(\alpha)} = (\boldsymbol{r}_{\boldsymbol{b},1}^{(\alpha)}, \ldots, \boldsymbol{r}_{\boldsymbol{b},c}^{(\alpha)})$ for each $b = 0, 1$ and $\alpha = 1, \ldots, \kappa$, where each $\boldsymbol{r}_{\boldsymbol{b},\beta}^{(\alpha)}$ is chosen and shared by $S_{j,\beta}$.

- The local randomness of each server to the multiparty garbling process of the virtual servers.

3. The servers commit their inputs to the multiparty garbling process of the virtual servers' local circuits.

4. The parties emulating each virtual server V_j run a multiparty garbling process to garble the local circuits of V_j in Π_0.

5. The servers run a verification on the preprocessing and input data to check their validity. Then, each server chooses his own watchlist of virtual servers and lets the servers emulating each virtual server on the watchlist open all the committed inputs and generated transcripts during the computation of the virtual server. The server then verifies whether the virtual servers on his watchlist all perform computation correctly.

6. The parties reconstruct the garbled circuits together with the input labels of each virtual server to the evaluator P_{king}, P_{king} then evaluates the circuit and sends the outputs as in Π_1'.

For more details, we refer the readers to Section J. Let $m = n$ and let each party serve as a client and a server at the same time, we give the following theorem in the standard MPC model.

Theorem 2. *Assuming random oracles and random OTs, for any constant $0 < \epsilon < 1$, there exists a computationally secure $(12 + R^{\mathsf{ROT}})$-round MPC protocol against a fully malicious adversary controlling up to $(1-\epsilon)n$ parties with communication of $O(|C|\kappa + D(n+\kappa)^2\kappa + n^3)$ bits plus $O(|C| + D(n+\kappa)^2)$ instances of ROT of message length $O(\kappa)$, where $|C|$ is the circuit size, D is the circuit depth, R^{ROT} is the number of rounds for an instance of ROT, and κ is the computational security 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.

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

References

[ABT19] Benny Applebaum, Zvika Brakerski, and Rotem Tsabary. Degree 2 is complete for the round-complexity of malicious mpc. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 504–531. Springer, 2019.

[ACGJ19] Prabhanjan Ananth, Arka Rai Choudhuri, Aarushi Goel, and Abhishek Jain. Two round information-theoretic mpc with malicious security. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 532–561. Springer, 2019.

[ACGJ20] Prabhanjan Ananth, Arka Rai Choudhuri, Aarushi Goel, and Abhishek Jain. Towards efficiency-preserving round compression in mpc. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 181–212. Springer, 2020.

[BCO+21] Aner Ben-Efraim, Kelong Cong, Eran Omri, Emmanuela Orsini, Nigel P. Smart, and Eduardo Soria-Vazquez. Large scale, actively secure computation from LPN and free-xor garbled circuits. In Anne Canteaut and 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.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.

[BFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference*, volume 7417 of *Lecture Notes in Computer Science*, pages 663–680. Springer, 2012.

[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.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12*, pages 784–796. ACM, 2012.

[BL18] Fabrice Benhamouda and Huijia Lin. k-round multiparty computation from k-round oblivious transfer via garbled interactive circuits. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, volume 10821 of *Lecture Notes in Computer Science*, pages 500–532. Springer, 2018.

[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 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security*, 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.

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

[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.

[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.

[CGH+18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64. Springer, 2018.

[CMOS25] Michele Ciampi, Ankit Kumar Misra, Rafail Ostrovsky, and Akash Shah. Black-box constant-round secure 2pc with succinct communication. In *Advances in Cryptology – EUROCRYPT 2025*, Lecture Notes in Computer Science. Springer, 2025.

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

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

[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.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[EGP+23] Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, Yifan Song, and Chenkai Weng. Superpack: Dishonest majority mpc with constant online communication. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 220–250. Springer, 2023.

[FR23] Thibauld Feneuil and Matthieu Rivain. Threshold linear secret sharing to the rescue of mpc-in-the-head. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023*, volume 14438 of *Lecture Notes in Computer Science*, pages 441–473. Springer, 2023.

[GIS+10] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In Daniele Micciancio, editor, *Theory of Cryptography*, pages 308–326. Springer, 2010.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, pages 39–56. Springer, 2008.

[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, Singapore, 2025. Springer Nature Singapore.

[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.

[GS96] Arnaldo Garcia and Henning Stichtenoth. On the asymptotic behaviour of some towers of function fields over finite fields. *Journal of Number Theory*, 61(2):248–273, 1996.

[GS18] Sanjam Garg and Akshayaram Srinivasan. Two-round multiparty secure computation from minimal assumptions. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 10821 of *Lecture Notes in Computer Science*, pages 468–499. Springer, 2018.

[HKN+25] David Heath, Vladimir Kolesnikov, Varun Narayanan, Rafail Ostrovsky, and Akash Shah. Multiparty garbling from OT with linear scaling and RAM support. Cryptology ePrint Archive, Paper 2025/444, 2025.

[HOSS18a] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, tinykeys for tinyot). In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security*, 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 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security*, 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, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[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.

[IKSS22] Yuval Ishai, Dakshita Khurana, Amit Sahai, and Akshayaram Srinivasan. Round-optimal black-box protocol compilers. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, volume 13275 of *Lecture Notes in Computer Science*, pages 210–240. Springer, 2022.

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

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. Flexor: Flexible garbling for XOR gates that beats free-xor. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference*, volume 8617 of *Lecture Notes in Computer Science*, pages 440–457. Springer, 2014.

[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*, volume 9215 of *Lecture Notes in Computer Science*, pages 724–741. Springer, 2015.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842. ACM, 2016.

[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.

[LOS14] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, volume 8617 of *Lecture Notes in Computer Science*, pages 495–512. Springer, 2014.

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

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In Stuart I. Feldman and Michael P. Wellman, editors, *Proceedings of the First ACM Conference on Electronic Commerce (EC-99)*, pages 129–139. ACM, 1999.

[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.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

[RR21] Mike Rosulek and Lawrence Roy. Three halves make a whole? beating the half-gates lower bound for garbled circuits. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021*, volume 12825 of *Lecture Notes in Computer Science*, pages 94–124. Springer, 2021.

[RS22] Rahul Rachuri and Peter Scholl. Le mans: Dynamic and fluid MPC for dishonest majority. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, volume 13507 of *Lecture Notes in Computer Science*, pages 719–749. Springer, 2022.

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

[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.

[ZLC+08] Zhifang Zhang, Mulan Liu, Yeow Meng Chee, San Ling, and Huaxiong Wang. Strongly multiplicative and 3-multiplicative linear secret sharing schemes. In Josef Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security*, volume 5350 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2008.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 9057 of *Lecture Notes in Computer Science*, pages 220–250. Springer, 2015.

A The Security Model

In the client-server model, clients provide inputs to the functionality and receive outputs, and servers can participate in the computation but do not have inputs or get outputs. Each party may have different roles in the computation. Note that, if every party plays a single client and a single server, this corresponds to a protocol in the standard MPC model. Let m denote the number of clients and n denote the number of servers. For all clients and servers, we assume that every two of them are connected via a secure (private and authentic) synchronous channel so that they can directly send messages to each other. The communication complexity is measured by the number of bits via private channels.

We consider an adversary that may corrupt t_c clients and t_s servers. In this work, we consider both semi-honest adversaries and fully malicious adversaries.

- If $\mathcal{A}$ is semi-honest, then corrupted clients and servers honestly follow the protocol.
- If $\mathcal{A}$ is fully malicious, then corrupted clients and servers can deviate from the protocol arbitrarily.

Real-World Execution. In the real world, the adversary $\mathcal{A}$ controlling corrupted clients and servers interacts with honest clients and servers. At the end of the protocol, the output of the real-world execution includes the inputs and outputs of honest clients and servers and the view of the adversary.

Ideal-World Execution. In the ideal world, a simulator Sim simulates honest clients and servers and interacts with the adversary $\mathcal{A}$. Furthermore, Sim has one-time access to $\mathcal{F}$, which includes providing inputs of corrupted clients and servers to $\mathcal{F}$, receiving the outputs of corrupted clients and servers, and sending instructions specified in $\mathcal{F}$. The output of the ideal-world execution includes the inputs and outputs of honest clients and servers and the view of the adversary.

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

Benefit of the Client-Server Model A benefit of the client-server model is that we only need to consider the maximal corruption of servers. At a high level, for an adversary $\mathcal{A}$ which controls $t'_s < t_s$ servers, we may construct another adversary $\mathcal{A}'$ which controls additional $t_s - t'_s$ servers and behaves as follows:

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

Note that, if a protocol is secure against $\mathcal{A}'$, then this protocol is also secure against $\mathcal{A}$ since the additional $t_s - t'_s$ parties controlled by $\mathcal{A}'$ honestly follow the protocol in both cases. Thus, we only need to focus on $\mathcal{A}'$ instead of $\mathcal{A}$.

B Basic Algebraic Geometry

In this part, we introduce some basic algebraic geometry used in the construction of a secret sharing scheme in [CC06].

Let C be a smooth, projective, absolutely irreducible curve defined over $\mathbb{F}_q$, and let g denote the genus of C. Let $\overline{\mathbb{F}_q}$ denote the algebraic closure of $\mathbb{F}_q$. A plane such curve can be represented by some polynomial $F(X, Y) \in \mathbb{F}_q[X, Y]$ that is irreducible in $\overline{\mathbb{F}_q}[X, Y]$. The affine part of the curve is defined as the set of points $P \in \overline{\mathbb{F}_q}^2$ such that $F(P) = 0$. By taking its projective closure, which amounts to introducing an extra variable, homogenizing the polynomial, and considering the zeroes in the two-dimensional projective space $\mathbb{P}^2(\overline{\mathbb{F}_q})$, one obtains the entire curve. More generally, curves defined over $\mathbb{F}_q$ is the "set of zeroes" in $\mathbb{P}^m(\overline{\mathbb{F}_q})$

of a homogeneous ideal $I \subset \mathbb{F}_q[X_0, \ldots, X_m]$, where I is such that its function field has transcendence degree 1 over the ground field, i.e., it is a one-dimensional variety. For a projective curve $V \subset \mathbb{P}^m(\overline{\mathbb{F}_q})$ defined by $m-1$ irreducible functions $f_1, \ldots, f_{m-1}$, if the at every point $P \in \overline{\mathbb{F}_q}^m$ in the affine part of V (such that $f_1(P) = \cdots = f_{m-1}(P) = 1$) is nonsingular, i.e. the $(m-1) \times m$ matrix containing the m partial derivatives of $f_1, \ldots, f_{m-1}$ has rank $m-1$, V is said to be smooth.

$\overline{\mathbb{F}_q}(C)$ denotes the function field of the curve. Very briefly, it consists of all fractions of polynomials $a, b \in \overline{\mathbb{F}_q}[X_0, \ldots, X_m]$, $b \notin I$, such that both are homogeneous of the same degree, under the equivalence relation that $a/b \sim a'/b'$ if $ab' \equiv a'b \bmod I$. The elements can be viewed as maps from the curve to $\overline{\mathbb{F}_q}$, and they have at most a finite number of poles and zeroes unless it is the zero function. Their "multiplicities add up to zero."

Since C is smooth at each point $P \in C$ by assumption, the local ring $\mathcal{O}_P(C)$ of functions $f \in \overline{\mathbb{F}_q}(C)$ that are well-defined at P (equivalently, the ones that do not have a pole at P) is a discrete valuation ring. Thus, at each $P \in C$, there exists $t \in \overline{\mathbb{F}_q}(C)$ (a uniformizing parameter) such that $t(P) = 0$ and each $f \in \mathcal{O}_P(C)$ can be uniquely written as $f = u \cdot t^{\nu_p(f)}$. Here, $u \in \mathcal{O}_P(C)$ is a unit (i.e. $u(P) \neq 0$), and $\nu_P(f)$ is a non-negative integer. This valuation ν_P extends to all of $\overline{\mathbb{F}_q}(C)$, by defining $\nu_P(f) = -\nu_P(1/f)$ if f has a pole at P.

A divisor is a formal sum $\sum_{P \in C} m_p \cdot (P)$ with integer coefficients m_p taken over all points P of the curve C. Divisors are required to have finite support, i.e., they are zero except possibly at finitely many points. The divisor of $f \in \mathbb{F}_q(C)$ is defined as $\mathsf{div}(f) = \sum_{P \in C} \nu_P(f) \cdot (P)$. It holds that $\mathsf{deg}(\mathsf{div}(f)) = 0$. The degree $\mathsf{deg}(D)$ of a divisor D is the sum $\sum_{P \in C} m_P \in \mathbb{Z}$ of its coefficients m_P.

The Riemann-Roch space associated with a divisor D is defined as $\mathcal{L}(D) = \{f \in \overline{\mathbb{F}_q}(C) | \mathsf{div}(f) + D \geq 0\} \cup \{0\}$. This is an $\overline{\mathbb{F}_q}$-vector space. The (partial) ordering "$\geq$" refers to the comparison of integer vectors and declaring one larger than the other if this holds coordinate-wise. Its dimension is denoted $\ell(D)$. This dimension is equal to 0 if $\mathsf{deg}(D) < 0$. The Riemann-Roch Theorem is concerned with the dimensions of those spaces. It says that $\ell(D) - \ell(K - D) = \mathsf{deg}(D) + 1 - g$. Here K is a canonical divisor. These are the divisors K of degree $2g-2$ and $\ell(K) = g$. It follows immediately that $\ell(D) = \mathsf{deg}(D) + 1 - g$ if $\mathsf{deg}(D)$ is at least $2g - 1$.

An $\mathbb{F}_q$-rational point on C is one whose projective coordinates can be chosen in $\mathbb{F}_q$. The set of $\mathbb{F}_q$-rational point on C is denoted by $C(\mathbb{F}_q)$. Below is a basic lemma in algebraic geometry.

C Security Proof for Protocol Π_1

We prove Theorem 3 as follows.

Proof. We prove the security of Π_1 by constructing an ideal adversary Sim_1. Then we will show that the output in the ideal world is computationally indistinguishable from that in the real world using hybrid arguments. Our simulation is in the client-server model where the adversary corrupts any number of clients and exactly t servers.

From the requirements of Π_0, there exists a PPT simulator Sim_0 that can generate the view of corrupted parties (where each party may be either a client or a server) together with honest parties' shares of $[\boldsymbol{s}_1]^{(2)}, \ldots, [\boldsymbol{s}_{\mathsf{rec}}]^{(2)}$ in Π_0 from corrupted clients' inputs and outputs.

Without loss of generality, we suppose that P_{king} is corrupted. We give the ideal adversary Sim_1 below.

Simulator Sim_1

Let $\mathcal{A}'$ be the adversary in Π_0 that behaves the same as $\mathcal{A}$ while interacting with $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ in the preprocessing step and the input step, and fail-stops corrupted parties before the evaluation phase. Sim_1 invokes Sim_0 with adversary $\mathcal{A}'$. When Sim_0 invokes $\mathcal{F}$, Sim_1 sends the same message to $\mathcal{F}$. Then, Sim_1 gets the output of Sim_0.

1. **Preprocessing.** Sim_1 simulates the preprocessing step of Π_1 as Π_0 with the output of Sim_0.
2. **Input.** Sim_1 simulates the input step of Π_1 as Π_0 with the output of Sim_0.

3. **Garbling Local Circuits.** For $i = 1, \ldots, \mathsf{rec}$, Sim_1 gets honest servers' shares of the sharing $[\boldsymbol{s}_i]^{(2)}$ from the output of Sim_0. Sim_1 then runs the algorithm Alg_2 in Remark 1 to find a $\Sigma^{(2)}$-sharing as $[\boldsymbol{s}_i]^{(2)}$ for each $i = 1, \ldots, \mathsf{rec}$ such that the honest servers' shares match the output of Sim_0. Then, if the receiver R_i of the i-th reconstruction is honest:
   (a) Sim_1 runs $\mathsf{Gen}(1^\kappa)$ k times to get keys $\boldsymbol{r}_{s_i,\beta} = (r^{(1)}_{s_i,\beta}, \ldots, r^{(\kappa)}_{s_i,\beta})$ for $\beta = 1, \ldots, k$. Let $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i} = (r^{(\alpha)}_{s_i,1}, \ldots, r^{(\alpha)}_{s_i,k})$ for each $\alpha = 1, \ldots, \kappa$.
   (b) For $\alpha = 1, \ldots, \kappa$, Sim_1 randomly samples corrupted servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$ (using the algorithm Alg_3 in Remark 1, same below) and sends them to the corrupted servers on behalf of R_i.
   (c) For $\alpha = 1, \ldots, \kappa$, Sim_1 randomly samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)}$ based on the corrupted servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}$.
   (d) For each honest server S_j, suppose that S_j's shares of $[\boldsymbol{s}_i]^{(2)}$ and $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)}$ are s_{i,S_j} and $r^{(\alpha)}_{s_i,S_j}$ respectively, Sim_1 runs $\mathsf{Sim}_{\mathsf{GC}}(1^\kappa, \mathtt{C}_i^{S_j}, (s_{i,S_j}, r^{(1)}_{s_i,S_j}, \ldots, r^{(\kappa)}_{s_i,S_j}))$ to obtain $(\mathtt{GC}_i^{S_j}, X_i^{S_j}, d_i^{S_j})$. Then Sim_1 sends $\mathtt{GC}_i^{S_j}, d_i^{S_j}$ to P_{king} on behalf of S_j. Here $\mathsf{Sim}_{\mathsf{GC}}$ is the PPT algorithm in definition 6 for the garbling scheme $\mathcal{G}$.

   If R_i is corrupted:
   (a) Sim_1 receives honest servers' shares of $[\boldsymbol{r}^{(\alpha)}_0]^{(3)}, [\boldsymbol{r}^{(\alpha)}_1 - \boldsymbol{r}^{(\alpha)}_0]$ from R_i for each $\alpha = 1, \ldots, \kappa$.
   (b) For each honest server S_j, suppose that S_j's shares of $[\boldsymbol{s}_i]^{(2)}$ and $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)}$ (computed with the shares of $[\boldsymbol{r}^{(\alpha)}_0]^{(3)}, [\boldsymbol{r}^{(\alpha)}_1 - \boldsymbol{r}^{(\alpha)}_0]$ received from R_i) are s_{i,S_j} and $r^{(\alpha)}_{s_i,S_j}$ respectively, Sim_1 runs $\mathsf{Sim}_{\mathsf{GC}}(1^\kappa, \mathtt{C}_i^{S_j}, (s_{i,S_j}, r^{(1)}_{s_i,S_j}, \ldots, r^{(\kappa)}_{s_i,S_j}))$ to obtain $(\mathtt{GC}_i^{S_j}, X_i^{S_j}, d_i^{S_j})$. Then Sim_1 sends $\mathtt{GC}_i^{S_j}, d_i^{S_j}$ to P_{king} on behalf of S_j.
4. **Encrypting Input Labels.** For each $i = 1, \ldots, \mathsf{rec}$, if the receiver R_i is a server and the β-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_β in R_i's circuit $\mathtt{Circ}_\gamma^{R_i}$ ($\gamma \in \{i+1, \ldots, \mathsf{rec}\}$), Sim_1 gets $X^{R_i}_{\gamma,j_\beta}$ from $X^{R_i}_\gamma$ and computes
$$\mathsf{ct}^{(i,\gamma)}_{j_\beta,s_{i,\beta}} = \mathsf{Enc}(\boldsymbol{r}_{s_i,\beta}, X^{R_i}_{\gamma,j_\beta}).$$
   Then, Sim_1 runs $\mathsf{Gen}(1^\kappa)$ to get a random key rk and encrypts an all-0 message m in the image space of $\mathsf{En}_{j_\beta}(e^{R_i}_\gamma, \cdot)$ by
$$\mathsf{ct}^{(i,\gamma)}_{j_\beta,1\oplus s_{i,\beta}} = \mathsf{Enc}(\mathsf{rk}, \mathsf{m}).$$
   Then, Sim_1 sends $\{\mathsf{ct}^{(i,\gamma)}_{j_\beta,0}, \mathsf{ct}^{(i,\gamma)}_{j_\beta,1}\}$ to P_{king} on behalf of R_i.
5. **Sending Input Labels.** For each honest server S_j and each $i = 1, \ldots, \mathsf{rec}$, Sim_1 gets the garbled input $X^{S_j}_{i,\gamma}$ from $X_i^{S_j}$ for each input wire value for the γ-th input wire of $\mathtt{C}_i^{S_j}$ that does not come from reconstruction. Then Sim_1 sends $X^{S_j}_{i,\gamma}$ to P_{king} on behalf of S_j.
6. **Sending Outputs.**
   (a) For each receiver R_i that is an honest client, Sim_1 receives $\boldsymbol{s}_i, \{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}\}_{\alpha=1}^\kappa$ from P_{king}.
   (b) For each receiver R_i that is an honest client, Sim_1 checks whether $\boldsymbol{s}_i, \{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}\}_{\alpha=1}^\kappa$ from P_{king} are all correctly sent. If not, Sim_1 aborts the protocol on behalf of R_i.
7. Sim_1 outputs the adversary's view.

Figure 8: The simulator for Π_1 when P_{king} is corrupted.

We construct the following hybrids:

Hyb$_0$: In this hybrid, Sim_1 gets honest clients' inputs and runs the protocol honestly. This corresponds to the real-world scenario.

Hyb$_1$: In this hybrid, for each $i = 1, \ldots, \mathsf{rec}$ with honest R_i and $\alpha = 1, \ldots, \kappa$, while generating the sharings $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$, Sim_1 first samples corrupted servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$ and then randomly samples honest servers' shares based on the corrupted servers' shares and the secret. Since Σ is an (n, t, k, ℓ)-LSSS that has 3-multiplicative reconstruction, each t shares of a Σ or $\Sigma^{(3)}$-sharing are uniformly random in $(\mathbb{F}_2^\ell)^t$ or $(\mathbb{F}_2^{\ell^3})^t$ respectively, so we only change the order of generating the honest servers' and the

corrupted servers' shares. Thus, $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ have the same output distribution.

$\mathbf{Hyb}_2$: In this hybrid, for each $i = 1, \ldots, \mathsf{rec}$ with an honest receiver R_i, Sim_1 additionally compute each honest server's share of $[\boldsymbol{s}_i]^{(2)}$ following Π_0. Then, Sim_1 runs the algorithm Alg_2 in Remark 1 with the honest servers' shares of $[\boldsymbol{s}_i]^{(2)}$ to decide the whole sharing $[\boldsymbol{s}_i]^{(2)}$ and uses them to compute corrupted servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$ for each $\alpha = 1, \ldots, \kappa$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

$\mathbf{Hyb}_3$: In this hybrid, for each $i = 1, \ldots, \mathsf{rec}$ with receiver R_i. If R_i is honest, Sim_1 doesn't generate the honest servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}$ first and then computes each honest server's share of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)}$ by himself. Instead, Sim_1 follows the protocol to generate $\boldsymbol{r}_{\boldsymbol{s}_i}$ first and then samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)}$ based on the corrupted servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)}$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}$. Then, Sim_1 computes the honest servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)}, [\boldsymbol{s}_i]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$. If R_i is corrupted, Sim_1 computes each honest server's shares of $[\boldsymbol{s}_i]^{(2)}$ and $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)}$ based on his shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$.

Moreover, Sim_1 doesn't follow the protocol to compute the garbled circuit by $\mathsf{Gb}(1^\kappa, \mathtt{C}_i^{S_j}) = (\mathtt{GC}_i^{S_j}, e_i^{S_j}, d_i^{S_j})$ and then sends $\mathtt{GC}_i^{S_j}, d_i^{S_j}$ to P_{king} on behalf of each honest server S_j. Instead, Sim_1 obtains $(\mathtt{GC}_i^{S_j}, X_i^{S_j}, d_i^{S_j})$ by running $\mathsf{Sim}_{\mathsf{GC}}(1^\kappa, \mathtt{C}_i^{S_j}, (s_{i,S_j}, r^{(1)}_{s_i,S_j}, \ldots, r^{(\kappa)}_{s_i,S_j}))$ and sends $\mathtt{GC}_i^{S_j}, d_i^{S_j}$ to P_{king}. Besides, while sending input labels to P_{king}, Sim_1 doesn't follow the protocol to compute $\mathsf{En}_\gamma(e_i^{S_j}, x_\gamma)$ for each input wire value x_γ of the γ-th input wire of $\mathtt{C}_i^{S_j}$. Instead, Sim_1 gets each input label $X^{S_j}_{i,\gamma}$ directly from $X_i^{S_j}$. In addition, for each $\beta = 1, \ldots, k$ and each R_i that is an honest server, if the β-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_β in R_i's circuit $\mathtt{Circ}_\gamma^{R_i}$, the ciphertext $\mathsf{ct}^{(i,\gamma)}_{j_\beta, 1\oplus s_{i,\beta}}$ is not computed by following the protocol. Instead, Sim_1 generates a random key with Gen and encrypts an all-0 message m in the image space of $\mathsf{En}_{j_\beta}(e_\gamma^{R_i}, \cdot)$. To prove that the distributions of $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ are computationally indistinguishable, we additionally construct the following hybrids between $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_3$.

$\mathbf{Hyb}_{3.1.1}$: In this hybrid, Sim_1 additionally computes each honest server S_j's share of $[\boldsymbol{s}_1]^{(2)}$ by using the input labels associated with the input of S_j to evaluate the garbled circuit $\mathtt{GC}_1^{S_j}$ with Ev. We denote the result of the computation be $\overline{[\boldsymbol{s}_1]^{(2)}}$. Since the input of $\mathtt{Circ}_1^{S_j}$ completely comes from the output of $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$, and the computation process of S_j's share of $[\boldsymbol{s}_1]^{(2)}$ in Π_0 is identical to $\mathtt{Circ}_1^{S_j}$, from the correctness condition of the garbling scheme, the result $\overline{[\boldsymbol{s}_1]^{(2)}}$ is the same as $[\boldsymbol{s}_1]^{(2)}$ from the execution of Π_0. If R_1 is honest, Sim_1 doesn't generate the honest servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}$ based on the corrupted servers' shares first and then compute each honest server's share of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1}]^{(3)}$ by himself. Instead, Sim_1 generates $\boldsymbol{r}_{\boldsymbol{s}_1}$ first and then samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1}]^{(3)}$ based on the corrupted servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1}]^{(3)}$ and the secret. Then, Sim_1 computes the honest servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1}]^{(3)}, [\boldsymbol{s}_1]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$. If R_1 is corrupted, Sim_1 still computes each honest server's shares of $[\boldsymbol{s}_1]^{(2)}$ and $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1}]^{(3)}$ based on his shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$. Since the honest servers' shares of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}$ are sampled randomly based on corrupted servers' shares and the secret in the last hybrid, their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)} + [\boldsymbol{s}_1]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$ are also random based on corrupted servers' shares and the secret. Therefore, we only change the order of generating honest servers' shares of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}$ and $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1}]^{(3)}$ without changing their distributions. Thus, $\mathbf{Hyb}_{3.1.1}$ and $\mathbf{Hyb}_2$ have the same distribution.

$\mathbf{Hyb}_{3.1.2}$: In this hybrid, Sim_1 doesn't follow the protocol to compute the garbled circuit by $\mathsf{Gb}(1^\kappa, \mathtt{C}_1^{S_j}) = (\mathtt{GC}_1^{S_j}, e_1^{S_j}, d_1^{S_j})$ and then sends $\mathtt{GC}_1^{S_j}, d_1^{S_j}$ to P_{king} on behalf of each honest server S_j. Instead, Sim_1 runs $\mathsf{Sim}_{\mathsf{GC}}(1^\kappa, \mathtt{C}_1^{S_j}, (s_{1,S_j}, r^{(1)}_{s_1,S_j}, \ldots, r^{(\kappa)}_{s_1,S_j}))$ to obtain $(\mathtt{GC}_1^{S_j}, X_1^{S_j}, d_1^{S_j})$ and sends $\mathtt{GC}_1^{S_j}, d_1^{S_j}$ to P_{king}. Besides, while sending input labels to P_{king}, Sim_1 doesn't follow the protocol to compute $\mathsf{En}_\gamma(e_1^{S_j}, x_\gamma)$ for each input wire value x_γ of the γ-th input wire of $\mathtt{C}_1^{S_j}$. Instead, Sim_1 gets each input label $\mathsf{En}_\gamma(e_1^{S_j}, x_\gamma)$ directly from $X_1^{S_j}$. From the definition of the private garbling scheme (definition 6), the distributions of the tuple $(\mathtt{GC}_1^{S_j}, d_1^{S_j})$ computed in the two different ways are computationally indistinguishable if $s_{1,S_j}, r^{(1)}_{s_1,S_j}, \ldots, r^{(\kappa)}_{s_1,S_j}$ are just the output of $\mathtt{C}_1^{S_j}$. Since the output of S_j's share of $[\boldsymbol{s}_1]^{(2)}$ by $\mathtt{C}_1^{S_j}$ is the same as the output of S_j's share of

$[\boldsymbol{s}_i]^{(2)}$ by $\mathtt{Circ}_1^{S_j}$, which is the same as $\boldsymbol{s}_1$ in Π_0. Besides, the output of S_j's shares $(\overline{r^{(1)}_{s_1,S_j}}, \ldots, \overline{r^{(\kappa)}_{s_1,S_j}})$ from $\mathtt{C}_1^{S_j}$ are computed by

$$[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)} + [\boldsymbol{s}_1]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}], \quad \alpha = 1, \ldots, \kappa,$$

which has the same distribution as the shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1}]^{(3)}$ we generated in the last hybrid. Therefore, the distributions of $\mathsf{En}_\gamma(e_1^{R_1}, x_\gamma)$ and the corresponding garbled input obtained from $X_1^{R_1}$ are also computationally indistinguishable. Thus, the distributions of $\mathbf{Hyb}_{3.1.2}$ and $\mathbf{Hyb}_{3.1.1}$ are computationally indistinguishable.

Note that for each $\beta = 1, \ldots, k$, if the β-th bit of $\boldsymbol{s}_1$ is 0, then we don't need the key $\boldsymbol{r}_{1,\beta}$ associated with the β-th bit of $\boldsymbol{s}_1$ since it is not used to generate any transcript sent to $\mathcal{A}$ in Step 3.(a)-(d), Sim_1 delays the generation of $\boldsymbol{r}_{1,\beta}$ when it is needed in the encryption in Step 4 in future hybrids. Similarly, if the β-th bit of $\boldsymbol{s}_1$ is 1, Sim_1 delays the generation of $\boldsymbol{r}_{0,\beta}$ when it is needed in the encryption in Step 4 in future hybrids.

$\mathbf{Hyb}_{3.2.1}$: In this hybrid, for each $\beta = 1, \ldots, k$ and $\eta = 2, \ldots, \mathsf{rec}$, if R_1 is an honest server and $\mathtt{C}_\eta^{R_1}$ takes the β-th bit of $\boldsymbol{s}_1$ as an input for the a-th input wires, the ciphertext $\mathsf{ct}^{(1,\eta)}_{a,1\oplus s_{1,\beta}}$ is not computed by following the protocol. Instead, Sim_1 generates a random key with Gen and encrypts an all-0 message m in the image space of $\mathsf{En}_a(e_\eta^{S_j}, \cdot)$. From the definition of a symmetric key scheme, for all security parameters κ and every choice of vectors $(m_0^{(1)}, \ldots, m_0^{(q)})$ and $(m_1^{(1)}, \ldots, m_1^{(q)})$, where $q = \mathsf{poly}(\kappa)$, it holds that:

$$\{\{\mathsf{Enc}(k, m_0^{(i)})\}_{i=1}^q : k \leftarrow \mathsf{Gen}(1^\kappa)\} \equiv_c \{\{\mathsf{Enc}(k, m_1^{(i)})\}_{i=1}^q : k \leftarrow \mathsf{Gen}(1^\kappa)\}.$$

If the β-th bit of $\boldsymbol{s}_1$, $s_{i,\beta}$ is equal to 0, we take $(m_0^{(1)}, \ldots, m_0^{(q)})$ to be the vector of $\mathsf{En}_a(e_\eta^{S_j}, 1)$ for all the circuit $\mathtt{C}_\eta^{R_1}$ takes the β-th bit of $\boldsymbol{s}_1$ as an input, and $(m_0^{(1)}, \ldots, m_0^{(q)})$ to be the all-0 vector of the same length. Since Sim generates $\boldsymbol{r}_{1,\beta}$ by $\mathsf{Gen}(1^\kappa)$ while doing the encryption, we know that the joint distribution of $\mathsf{Enc}(\boldsymbol{r}_{1,\beta}, \mathsf{En}_a(e_\eta^{S_j}, 1))$ for all these circuits $\mathtt{C}_\eta^{R_1}$ (that takes the β-th bit of $\boldsymbol{s}_1$ as input) is computationally indistinguishable from the joint distribution of $\mathsf{ct}^{(1,\eta)}_{a,1\oplus s_{i,\beta}} = \mathsf{Enc}(\mathsf{rk}, \mathsf{m})$ for all these circuits. Similarly, $\mathsf{Enc}(\boldsymbol{r}_{0,\beta}, \mathsf{En}_a(e_\eta^{S_j}, 0))$ is computationally indistinguishable from $\mathsf{ct}^{(1,\eta)}_{a,1\oplus s_{1,\beta}} = \mathsf{Enc}(\mathsf{rk}, \mathsf{m})$ when $s_{1,\beta} = 1$. Thus, the distributions of $\mathbf{Hyb}_{3.2.1}$ and $\mathbf{Hyb}_{3.1.2}$ are computationally indistinguishable.

$\mathbf{Hyb}_{3.2.2}$: In this hybrid, Sim_1 additionally computes each honest server S_j's share of $[\boldsymbol{s}_2]^{(2)}$ by using the input labels associated with the input of S_j to evaluate the garbled circuit $\mathtt{GC}_2^{S_j}$ with Ev. We denote the result of the computation be $\overline{[\boldsymbol{s}_2]^{(2)}}$. Since the input of $\mathtt{Circ}_2^{S_j}$ completely comes from the output of $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the reconstruction of $\overline{[\boldsymbol{s}_1]^{(2)}}$, and since $\overline{[\boldsymbol{s}_1]^{(2)}} = [\boldsymbol{s}_1]^{(2)}$ and the computation process of S_j's share of $[\boldsymbol{s}_2]^{(2)}$ is the same in Π_0 with $\mathtt{Circ}_2^{S_j}$, from the correctness condition of the garbling scheme, the result $\overline{[\boldsymbol{s}_2]^{(2)}}$ is the same as $[\boldsymbol{s}_2]^{(2)}$ from the execution of Π_0. If R_2 is honest, Sim_1 doesn't generate the honest servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}$ first and then computes each honest server's share of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_2}]^{(3)}$ by himself. Instead, Sim_1 generates $\boldsymbol{r}_{\boldsymbol{s}_2}$ first and then samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_2}]^{(3)}$ based on the corrupted servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_2}]^{(3)}$ and the secret. Then, Sim_1 computes the honest servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_2}]^{(3)}, [\boldsymbol{s}_2]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$. For the same reason as in $\mathbf{Hyb}_{3.1.1}$, $\mathbf{Hyb}_{3.2.2}$ and $\mathbf{Hyb}_{3.2.1}$ have the same distribution.

$\mathbf{Hyb}_{3.2.3}$: In this hybrid, Sim_1 doesn't follow the protocol to compute the garbled circuit by $\mathsf{Gb}(1^\kappa, \mathtt{C}_2^{S_j}) = (\mathtt{GC}_2^{S_j}, e_2^{S_j}, d_2^{S_j})$ and then sends $\mathtt{GC}_2^{S_j}, d_2^{S_j}$ to P_{king} on behalf of each honest server S_j. Instead, Sim_1 runs $\mathsf{Sim}_{\mathsf{GC}}(1^\kappa, \mathtt{C}_2^{S_j}, (s_{2,S_j}, r^{(1)}_{s_2,S_j}, \ldots, r^{(\kappa)}_{s_2,S_j}))$ to obtain $(\mathtt{GC}_2^{S_j}, X_2^{S_j}, d_2^{S_j})$ and sends $\mathtt{GC}_2^{S_j}, d_2^{S_j}$ to P_{king}. Besides, while sending input labels to P_{king}, Sim_1 doesn't follow the protocol to compute $\mathsf{En}_\gamma(e_2^{S_j}, x_\gamma)$ for each input wire value x_γ of the γ-th input wire of $\mathtt{C}_2^{S_j}$ that does not come from reconstructions. Instead, Sim_1 gets each input label $\mathsf{En}_\gamma(e_2^{S_j}, x_\gamma)$ directly from $X_2^{S_j}$. For the same reason as in $\mathbf{Hyb}_{3.1.2}$, the distributions of $\mathbf{Hyb}_{3.2.3}$ and $\mathbf{Hyb}_{3.2.2}$ are computationally indistinguishable.

Note that for each $\beta = 1, \ldots, k$, if the β-th bit of $\boldsymbol{s}_2$ is 0, then we don't need the key $\boldsymbol{r}_{1,\beta}$ associated with this bit of $\boldsymbol{s}_2$ is not used to generate any transcript sent to $\mathcal{A}$ in Step 3.(a)-(d), Sim_1 delays the generation

of $\boldsymbol{r}_{1,\beta}$ when it is needed in the encryption in Step 4 in future hybrids. Similarly, if the β-th bit of $\boldsymbol{s}_2$ is 1, Sim_1 delays the generation of $\boldsymbol{r}_{0,\beta}$ when it is needed in the encryption in Step 4 in future hybrids.

Similarly, for each $\gamma = 3, \ldots, \mathsf{rec}$ we can define $\mathbf{Hyb}_{3.\gamma.1}, \mathbf{Hyb}_{3.\gamma.2}, \mathbf{Hyb}_{3.\gamma.3}$.

In $\mathbf{Hyb}_{3.\gamma.1}$, for each $\beta = 1, \ldots, k$ and $\eta = \gamma, \ldots, \mathsf{rec}$, if $R_{\gamma-1}$ is an honest server, $\mathtt{C}_\eta^{R_i}$ takes the β-th bit of $\boldsymbol{s}_i$ as an input for the a-th input wires, the ciphertext $\mathsf{ct}_{a,1\oplus s_{i,\beta}}^{(i,\eta)}$ is not computed by following the protocol. Instead, Sim_1 generates a random key with Gen and encrypts an all-0 message m in the image space of $\mathsf{En}_a(e_\gamma^{S_j}, \cdot)$.

In $\mathbf{Hyb}_{3.\gamma.2}$, Sim_1 additionally computes each honest server S_j's share of $[\boldsymbol{s}_\gamma]^{(2)}$ by following the evaluation process of P_{king} to evaluate $\mathtt{GC}_\gamma^{S_j}$ with Ev. We denote the result of the computation be $\overline{[\boldsymbol{s}_\gamma]^{(2)}}$. If R_γ is honest, Sim_1 doesn't generate the honest servers' shares of $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}$ first and then computes each honest server's share of each $[\boldsymbol{r}_{\boldsymbol{s}_\gamma}^{(\alpha)}]^{(3)}$ by himself. Instead, Sim_1 generates $\boldsymbol{r}_{\boldsymbol{s}_1}$ first and then samples the whole sharing $[\boldsymbol{r}_{\boldsymbol{s}_\gamma}^{(\alpha)}]^{(3)}$ based on the corrupted servers' shares of $[\boldsymbol{r}_{\boldsymbol{s}_\gamma}^{(\alpha)}]^{(3)}$ and the secret. Then, Sim_1 computes the honest servers' shares of $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}$ based on their shares of $[\boldsymbol{r}_{\boldsymbol{s}_\gamma}^{(\alpha)}]^{(3)}, [\boldsymbol{s}_\gamma]^{(2)}, [\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]$. If R_γ is corrupted, Sim_1 still computes each honest server's shares of $[\boldsymbol{s}_\gamma]^{(2)}$ and $[\boldsymbol{r}_{\boldsymbol{s}_\gamma}^{(\alpha)}]^{(3)}$ based on his shares of $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]$.

In $\mathbf{Hyb}_{3.\gamma.3}$, Sim_1 doesn't follow the protocol to compute the garbled circuit by $\mathsf{Gb}(1^\kappa, \mathtt{C}_\gamma^{S_j}) = (\mathtt{GC}_\gamma^{S_j}, e_\gamma^{S_j}, d_\gamma^{S_j})$ and then sends $\mathtt{GC}_\gamma^{S_j}, d_\gamma^{S_j}$ to P_{king} on behalf of each honest server S_j. Instead, Sim_1 obtains $(\mathtt{GC}_\gamma^{S_j}, X_\gamma^{S_j}, d_\gamma^{S_j})$ by running $\mathsf{Sim}_{\mathsf{GC}}(1^\kappa, \mathtt{C}_\gamma^{S_j}, (s_{\gamma,S_j}, r_{s_\gamma,S_j}^{(1)}, \ldots, r_{s_\gamma,S_j}^{(\kappa)}))$ and sends $\mathtt{GC}_\gamma^{S_j}, d_\gamma^{S_j}$ to P_{king}. Besides, while sending input labels to P_{king}, Sim_1 doesn't follow the protocol to compute $\mathsf{En}_\eta(e_\gamma^{S_j}, x_\eta)$ for each input wire value x_η of the η-th input wire of $\mathtt{C}_\gamma^{S_j}$ that does not come from reconstructions. Instead, Sim_1 gets each input label $\mathsf{En}_\eta(e_\gamma^{S_j}, x_\eta)$ directly from $X_\gamma^{S_j}$.

For the same reason as in $\mathbf{Hyb}_{3.2.1}, \mathbf{Hyb}_{3.2.2}, \mathbf{Hyb}_{3.2.3}$, for each $\gamma = 3, \ldots, \mathsf{rec}$, the distributions of $\mathbf{Hyb}_{3.\gamma.1}$ and $\mathbf{Hyb}_{3.(\gamma-1).3}$ are computationally indistinguishable, the distributions of $\mathbf{Hyb}_{3.\gamma.2}$ and $\mathbf{Hyb}_{3.\gamma.1}$ are computationally indistinguishable, and the distributions of $\mathbf{Hyb}_{3.\gamma.3}$ and $\mathbf{Hyb}_{3.\gamma.2}$ are also computationally indistinguishable.

Note that $\mathbf{Hyb}_{3.\mathsf{rec}.3}$ is just $\mathbf{Hyb}_3$, we conclude that $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same distribution.

Note that if R_i is an honest client, the outputs of k executions of Gen that are used to compute $\{\boldsymbol{r}_{\boldsymbol{1}\oplus\boldsymbol{s}_i}^{(\alpha)}\}_{\alpha=1}^\kappa$ are not used until R_i receives $\boldsymbol{s}_i, \{\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}\}_{\alpha=1}^\kappa$ and does the verification on them. Sim_1 delays the generation of $\{\boldsymbol{r}_{\boldsymbol{1}\oplus\boldsymbol{s}_i}^{(\alpha)}\}_{\alpha=1}^\kappa$ after R_i receives $\boldsymbol{s}_i, \{\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}\}_{\alpha=1}^\kappa$ in future hybrids.

$\mathbf{Hyb}_4$: In this hybrid, for each $i = 1, \ldots, \mathsf{rec}$ with honest receiver R_i who is a client, Sim_1 doesn't follow the protocol to check the values $\boldsymbol{s}_i, \{\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}\}_{\alpha=1}^\kappa$ received from P_{king}. Instead, Sim_1 checks whether they are correctly sent. Note that when they are correctly sent, then $\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)} = \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)} + \boldsymbol{s}_i * (\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)})$ must hold for each $\alpha = 1, \ldots, \kappa$. Thus, the output only changes when $\boldsymbol{s}_i, \{\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}\}_{\alpha=1}^\kappa$ are not correctly sent but for each $\alpha = 1, \ldots, \kappa$ it still holds that $\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)} = \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)} + \boldsymbol{s}_i * (\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)})$ (by the values received from P_{king}). Since when $\boldsymbol{s}_i$ is correctly sent, $\{\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}\}_{\alpha=1}^\kappa$ is determined by the equation $\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)} = \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)} + \boldsymbol{s}_i * (\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)})$, the output only changes when a bit of $\boldsymbol{s}_i$ is not correctly sent. Assume that it's the β-th bit. Note that if the β-th bit of $\boldsymbol{s}_i$ is 0, then the β-th bit of $\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}$ received from P_{king} should be the β-th bit of $\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)}$. These bits form the vector $\boldsymbol{r}_{\boldsymbol{1},\beta}$, which is generated by $\mathsf{Gen}(1^\kappa)$ after $\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}, \boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)}$ are received from P_{king}. Similarly, if the β-th bit of $\boldsymbol{s}_i$ is 1, then the β-th bit of $\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}$ received from P_{king} should be the β-th bit of $\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}$, and these bits form the vector $\boldsymbol{r}_{\boldsymbol{1},\beta}$ which is generated by $\mathsf{Gen}(1^\kappa)$ after $\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}, \boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)}$ are received from P_{king}. Thus, the output changes only when an output of $\mathsf{Gen}(1^\kappa)$ is guessed correctly by $\mathcal{A}$, and the probability is negligible (or it breaks the security of the symmetric key encryption scheme). Thus, the distributions of $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ are statistically close.

Note that if R_i is an honest client, we only need $\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}$ and we don't need $\boldsymbol{r}_{\boldsymbol{1}\oplus\boldsymbol{s}_i}^{(\alpha)}$ for the simulation, and we also don't need honest servers' shares of $\{[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]\}_{\alpha=1}^\kappa$ in the simulation. Sim_1 doesn't generate them in future hybrids.

$\mathbf{Hyb}_5$: In this hybrid, since all the transcripts between honest and corrupted parties generated by Sim_1

can be generated from the transcripts between honest and corrupted parties obtained in the execution of Π_0, Sim_1 just runs Π_0 first to obtain all the transcripts and then uses them to generate the output of Sim_1. In addition, honest clients don't follow the protocol Π_1 to compute their output. Instead, they follow Π_0 to get their output. Since the value $\boldsymbol{s}_i$ sent from P_{king} to each honest client in Π_1 is the same as the value $\boldsymbol{s}_i$ in Π_0, and the preprocessing and input data of Π_0, Π_1 is also the same, the computation of honest clients' output is in the two protocols is completely the same. Therefore, we only change the way of generating the output of Sim_1 without changing their distributions. Thus, $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same distribution.

$\mathbf{Hyb}_6$: In this hybrid, Sim_1 doesn't run Π_0 to get the transcripts between honest and corrupted parties in Π_0 and use them to generate all the transcripts between honest and corrupted parties in Π_1. Instead, Sim_1 invokes Sim_0 with $\mathcal{A}'$ to get the transcripts between honest and corrupted parties in Π_0. In addition, honest clients get their outputs from $\mathcal{F}$ instead of following Π_0 to compute them. From the requirements of Π_0, the joint distribution of transcripts between honest and corrupted parties in Π_0 and the honest clients' output in Π_0 is computationally indistinguishable from the joint distribution of the output of Sim_0 and honest clients' output from $\mathcal{F}$. Thus, the distributions of $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ are computationally indistinguishable.

Note that $\mathbf{Hyb}_6$ is the ideal-world scenario, Π_1 computes $\mathcal{F}$ with computational security.

□

D Security Proof for Protocol Π_1'

We prove Theorem 4 as follows.

Proof. We prove the security of Π_1' by constructing an ideal adversary Sim_1'. Then we will show that the output in the ideal world is computationally indistinguishable from that in the real world using hybrid arguments. Our simulation is in the client-server model where the adversary corrupts any number of clients and exactly t servers.

From the requirements of Π_0, there exists a PPT simulator Sim_0 that can generate all the transcripts between honest parties and corrupted parties in Π_0 from corrupted clients' inputs and outputs together with honest servers' shares of the sharings $[\boldsymbol{s}_1]^{(2)}, \ldots, [\boldsymbol{s}_{\mathsf{rec}}]^{(2)}$.

Without loss of generality, we suppose that P_{king} is corrupted. We give the ideal adversary Sim_1' below.

Simulator Sim_1'

Let $\mathcal{A}'$ be the adversary in Π_0 that behaves the same as $\mathcal{A}$ while interacting with $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ in the preprocessing step and the input step, and fail-stops all corrupted parties before the evaluation phase. Sim_1' invokes Sim_0 with adversary $\mathcal{A}'$. When Sim_0 invokes $\mathcal{F}$, Sim_1' sends the same message to $\mathcal{F}$. Then, Sim_1' gets the output of Sim_0.

1. **Preprocessing.** Sim_1' simulates the preprocessing step of Π_1' as Π_0 with the output of Sim_0.
2. **Input.** Sim_1' simulates the input step of Π_1' as Π_0 with the output of Sim_0.
3. **Generating Output Labels.** For $i = 1, \ldots, \mathsf{rec}$, Sim_1' gets honest servers' shares of the sharing $[\boldsymbol{s}_i]^{(2)}$ from the view of the adversary in Π_0. Sim_1' then runs the algorithm Alg_2 in Remark 1 to find a $\Sigma^{(2)}$-sharing as $[\boldsymbol{s}_i]^{(2)}$ for each $i = 1, \ldots, \mathsf{rec}$ such that the honest servers' shares match the output of Sim_0. Then:
   - If the receiver R_i of the i-th reconstruction is honest:
     (a) Sim_1' samples k random κ-bit string as $\boldsymbol{r}_{s_{i,\beta},\beta} = (r^{(1)}_{s_{i,\beta},\beta}, \ldots, r^{(\kappa)}_{s_{i,\beta},\beta})$ for $\beta = 1, \ldots, k$. Let $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i} = (r^{(\alpha)}_{s_{i,1},1}, \ldots, r^{(\alpha)}_{s_{i,k},k})$ for each $\alpha = 1, \ldots, \kappa$.
     (b) For $\alpha = 1, \ldots, \kappa$, Sim_1' randomly samples corrupted servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$ (using the algorithm Alg_3 in Remark 1, same below) and sends them to the corrupted servers on behalf of R_i.
     (c) For $\alpha = 1, \ldots, \kappa$, Sim_1' randomly samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)}$ based on the corrupted servers' shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}$.

(d) For each honest server S_j and each $a = 1, \ldots, \ell^2$, let $s_{i,a}^{S_j}$ be the a-th bit of S_j's share of $[\boldsymbol{s}_i]$ and

$$Y^{S_j}_{(i-1)\ell^2+a, s_{i,a}^{S_j}} = \big(([\boldsymbol{r}^{(1)}_{\boldsymbol{s}_i}]^{(3)})^{S_j}_{[a\ell+1,(a+1)\ell]}, \ldots, ([\boldsymbol{r}^{(\kappa)}_{\boldsymbol{s}_i}]^{(3)})^{S_j}_{[a\ell+1,(a+1)\ell]}\big).$$

– If R_i is corrupted:

(a) Sim'_1 receives honest servers' shares of $[\boldsymbol{r}_0^{(\alpha)}]^{(3)}, [\boldsymbol{r}_1^{(\alpha)} - \boldsymbol{r}_0^{(\alpha)}]$ from R_i for each $\alpha = 1, \ldots, \kappa$.

(b) For each honest server S_j and each $a = 1, \ldots, \ell^2$, let $s_{i,a}^{S_j}$ be the a-th bit of S_j's share of $[\boldsymbol{s}_i]$, Sim'_1 follows the protocol to compute $Y^{S_j}_{(i-1)\ell^2+a, s_{i,a}^{S_j}}$ based on S_j's shares of $\{[\boldsymbol{r}_0^{(\alpha)}]^{(3)}, [\boldsymbol{r}_1^{(\alpha)} - \boldsymbol{r}_0^{(\alpha)}]\}_{\alpha=1}^{\kappa}$ received from R_i.

4. **Garbling Local Circuits.** For each honest server S_j:

(a) For each wire w in $\mathtt{Circ}^{S_j}$ that is not an output wire of an XOR gate or an output gate, Sim'_1 samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string $k_{w,v_w\oplus\lambda_w}$.

(b) For each XOR gate in $\mathtt{Circ}^{S_j}$ with input wires a, b and output wire o, Sim'_1 computes $k_{o,v_o\oplus\lambda_o} = k_{a,v_a\oplus\lambda_a} \oplus k_{b,v_b\oplus\lambda_b}$ 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 $\mathtt{Circ}^{S_j}$ with input wire a, b and output wire o, Sim'_1 computes

$$\begin{aligned}&\mathsf{ct}^{(g)}_{v_a\oplus\lambda_a, v_b\oplus\lambda_b}\\ =&\mathcal{O}_1(k_{a,v_a\oplus\lambda_a}\|(v_a\oplus\lambda_a)\|k_{b,v_b\oplus\lambda_b}\|(v_b\oplus\lambda_b)\|g)\oplus k_{o,v_o\oplus\lambda_o}\|(v_o\oplus\lambda_o).\end{aligned}$$

Then Sim'_1 samples 3 random κ-bit strings as the other 3 ciphertexts for this gate g.

(d) For each input wire w of an output gate in $\mathtt{Circ}^{S_j}$, the output gate outputs a bit of S_j's share of a $\Sigma^{(2)}$-sharing that needs reconstruction in Π_0, which can be obtained from the output of Sim_0. Sim'_1 sets the output wire value v_w to be the corresponding bit from the output of Sim_0. Then, Sim'_1 computes $\lambda_w = (v_w \oplus \lambda_w) \oplus \lambda_w$.

(e) For each output gate in $\mathtt{Circ}^{S_j}$ indexed $1, \ldots, \ell^2\mathsf{rec}$ with input wire w, Sim'_1 computes

$$\mathsf{ct}_{w,v_w\oplus\lambda_w} = \mathcal{O}_2(k_{w,v_w\oplus\lambda_w}\|(v_w\oplus\lambda_w)\|w)\oplus Y_{k,v_w}.$$

Then Sim'_1 samples a random $\ell\kappa$-bit string as the other ciphertext for this wire w.

(f) Let the $\mathtt{GC}^{S_j}$ be the set of all the ciphertexts. Let d^{S_j} be the set of λ_w for each input wire w of output gates in $\mathtt{Circ}^{S_j}$. Sim'_1 sends $\mathtt{GC}^{S_j}, d^{S_j}$ to P_{king} on behalf of S_j.

5. **Encrypting Input Labels.** For $i = 1, \ldots, \mathsf{rec}$, if R_i is an honest server and the β-th bit of $\boldsymbol{s}_i$ is used as an input wire w_{j_β} with index j_β in R_i's circuits $\mathtt{Circ}^{R_i}$, Sim'_1 computes

$$\mathsf{ct}^{(i)}_{j_\beta, s_{i,\beta}} = \mathcal{O}_1(\boldsymbol{r}_{s_{i,\beta},\beta}\|s_{i,\beta}\|i\|\beta\|j_\beta)\oplus\big(k_{w_{j_\beta},v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}}}\|(v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}})\big).$$

Then, Sim'_1 samples a random κ-bit string as $\mathsf{ct}^{(i)}_{j_\beta,1\oplus s_{i,\beta}}$. Then, Sim'_1 sends $\{\mathsf{ct}_{\beta,0}, \mathsf{ct}_{\beta,1}\}$ to P_{king} on behalf of R_i.

6. **Sending Input Labels.** For each honest server S_j, Sim'_1 gets the garbled input $X_\gamma^{S_j} = k_{w_\gamma,v_{w_\gamma}\oplus\lambda_{w_\gamma}}\|(v_{w_\gamma}\oplus\lambda_{w_\gamma})$ for each input wire (with index γ) w_γ of S_j's local circuits that does not come from reconstruction. Then Sim'_1 sends $X_\gamma^{S_j}$ to P_{king} on behalf of S_j.

7. **Sending Outputs.**

(a) For each receiver R_i that is an honest client, Sim'_1 receives $\boldsymbol{s}_i, \{\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}\}_{\alpha=1}^{\kappa}$ from P_{king}.

(b) For each receiver R_i that is an honest client, Sim'_1 checks whether $\boldsymbol{s}_i, \{\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}\}_{\alpha=1}^{\kappa}$ from P_{king} are all correctly sent. If not, Sim'_1 aborts the protocol on behalf of R_i.

8. Sim'_1 outputs the adversary's view.

Figure 9: The simulator for Π'_1 when P_{king} is corrupted.

We construct the following hybrids:

$\mathbf{Hyb}_0$: In this hybrid, Sim_1' gets honest clients' inputs and runs the protocol honestly. This corresponds to the real-world scenario.

$\mathbf{Hyb}_1$: In this hybrid, for each $i = 1, \ldots, \mathsf{rec}$ with honest R_i and $\alpha = 1, \ldots, \kappa$, while generating the sharings $[\boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\mathbf{1}}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]$, Sim_1' first samples corrupted servers' shares of $[\boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\mathbf{1}}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]$ and then randomly samples honest servers' shares based on the corrupted servers' shares and the secret. Since Σ is an (n, t, k, ℓ)-LSSS that has 3-multiplicative reconstruction, each t shares of a Σ or $\Sigma^{(3)}$-sharing are uniformly random in $(\mathbb{F}_2^{\ell})^t$ or $(\mathbb{F}_2^{\ell^3})^t$ respectively, so we only change the order of generating the honest servers' and the corrupted servers' shares. Thus, $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ have the same output distribution.

$\mathbf{Hyb}_2$: In this hybrid, for each wire w in each honest server S_j's local circuit $\mathtt{Circ}^{S_j}$, Sim_1' additionally follows the execution of Π_0 to compute the value v_w of w. Then, Sim_1' runs the algorithm Alg_2 in Remark 1 with the honest servers' shares of $[\boldsymbol{s}_i]^{(2)}$ for each $i = 1, \ldots, \mathsf{rec}$ with an honest receiver R_i to decide the whole sharing $[\boldsymbol{s}_i]^{(2)}$ and uses them to compute corrupted servers' shares of $[\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}]^{(3)} = [\boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}_{\mathbf{1}}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]$ for each $\alpha = 1, \ldots, \kappa$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

$\mathbf{Hyb}_3$: In this hybrid, for each honest server S_j, Sim_1' doesn't follow the protocol to garble S_j's local circuit $\mathtt{Circ}^{S_j}$. Instead:

1. For $i = 1, \ldots, \mathsf{rec}$, if R_i is honest, Sim_1' doesn't generate the honest servers' shares of $[\boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]^{(3)}$ first and then computes each honest server's share of each $[\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}]^{(3)}$ by himself. Instead, Sim_1' follows the protocol to generate $\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}$ first and then samples the whole sharing $[\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}]^{(3)}$ based on the corrupted servers' shares of $[\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}]^{(3)}$ and the secret $\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}$. Then, Sim_1' computes the honest servers' shares of $[\boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]^{(3)}$ based on their shares of $[\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}]^{(3)}, [\boldsymbol{s}_i]^{(2)}, [\boldsymbol{r}_{\mathbf{1}}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0}}^{(\alpha)}]$. Then Sim_1' computes each $Y^{S_j}_{(i-1)\ell^2+a, s_{i,a}^{S_j}}$ by
$$Y^{S_j}_{(i-1)\ell^2+a, s_{i,a}^{S_j}} = \big(([\boldsymbol{r}_{\boldsymbol{s}_i}^{(1)}]^{(3)})^{S_j}_{[a\ell+1,(a+1)\ell]}, \ldots, ([\boldsymbol{r}_{\boldsymbol{s}_i}^{(\kappa)}]^{(3)})^{S_j}_{[a\ell+1,(a+1)\ell]}\big).$$
For the other label $Y^{S_j}_{(i-1)\ell^2+a, 1\oplus s_{i,a}^{S_j}}$, Sim_1' still follows the protocol to compute it. If R_i is corrupted, Sim_1' still follows the protocol to compute the labels $Y_{(i-1)\ell^2+a,0}, Y_{(i-1)\ell^2+a,1}$ based on S_j's shares of $\{[\boldsymbol{r}_0^{(\alpha)}]^{(3)}, [\boldsymbol{r}_1^{(\alpha)} - \boldsymbol{r}_0^{(\alpha)}]\}_{\alpha=1}^{\kappa}$ received from R_i.

2. For each wire w in each $\mathtt{Circ}^{S_j}$ that is not an output wire of an XOR gate or an AND gate, Sim_1' samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string as $k_{w, v_w \oplus \lambda_w}$. Then Sim_1' computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$ for these wires. For each wire w in $\mathtt{Circ}^{S_j}$ that is not an output wire of an output gate, Sim_1' computes $k_{w, 1\oplus v_w \oplus \lambda_w} = k_{w, v_w \oplus \lambda_w} \oplus \Delta^{S_j}$ at the end of the garbling process, where Δ^{S_j} is generated after all the ciphertexts are generated.

3. Sim_1' maintains a set Q_1. For each AND gate g in $\mathtt{Circ}^{S_j}$ with input wire a, b, and for all $i_0, i_1 \in \{0,1\}$ such that $(i_0, i_1) \neq (v_a \oplus \lambda_a, v_b \oplus \lambda_b)$, when the honest server S_j computes $\mathsf{ct}^{(g)}_{i_0,i_1}$, Sim_1' checks whether the query to the random oracle $\mathcal{O}_1$ has been queried before. If true, Sim_1' aborts the simulation. Otherwise, Sim_1' adds the query to Q_1.

4. Sim_1' maintains a set Q_2. For each input wire w of an output gate in $\mathtt{Circ}^{S_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, when the honest server S_j computes ct_{w,i_2}, Sim_1' checks whether the query to the random oracle $\mathcal{O}_2$ has been queried before. If true, Sim_1' aborts the simulation. Otherwise, Sim_1' adds the query to Q_2.

5. For each AND gate g in $\mathtt{Circ}^{S_j}$ with input wire a, b and output wire o, Sim_1' doesn't follow the protocol to compute the ciphertexts except $\mathsf{ct}^{(g)}_{v_a\oplus\lambda_a, v_b\oplus\lambda_b}$. Instead, Sim_1' samples 3 random κ-bit strings as them. Sim_1' computes the output of $\mathcal{O}_1$ 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 $\mathtt{Circ}^{S_j}$ indexed $1, \ldots, f.m$ with input wire w, Sim'_1 doesn't follow the protocol to compute the ciphertext $\mathsf{ct}_{w,1\oplus v_w\oplus\lambda_w}$. Instead, Sim'_1 samples a random $\ell\kappa$-bit string as it. Sim'_1 computes the output of $\mathcal{O}_2$ to the queries that are used to generate these ciphertexts based on the random strings and the output labels.

7. For $i = 1, \ldots, \mathsf{rec}$, Sim'_1 doesn't honestly compute $\mathsf{En}_{j_\beta}(e^{S_j}, s_{i,\beta})$ for each input wire (with index γ) w_γ of $\mathtt{Circ}^{S_j}$ that does not come from reconstruction. Instead, Sim'_1 sets the garbled input $X_\gamma^{S_j}$ to be $k_{w_\gamma, v_{w_\gamma}\oplus\lambda_{w_\gamma}} \| (v_{w_\gamma} \oplus \lambda_{w_\gamma})$ for each w_γ.

8. For $i = 1, \ldots, \mathsf{rec}$, if S_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_β in $\mathtt{Circ}^{S_j}$, Sim'_1 doesn't follow the protocol computing $\mathsf{ct}^{(i)}_{j_\beta,1\oplus s_{i,\beta}}$. Instead, Sim'_1 samples a random κ-bit string as $\mathsf{ct}^{(i)}_{j_\beta,1\oplus s_{i,\beta}}$.

To prove that the distributions of $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ are computationally indistinguishable, we additionally construct the following hybrids between $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_3$.

$\mathbf{Hyb}_{3.0}$: In this hybrid, for each honest server S_j, while garbling the circuit $\mathtt{Circ}^{S_j}$, Sim'_1 garbles the sub-circuits $\mathtt{Circ}_1^{S_j}, \ldots, \mathtt{Circ}_{\mathsf{rec}}^{S_j}$ in order. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{3.0}$ and $\mathbf{Hyb}_2$ have the same distribution.

$\mathbf{Hyb}_{3.1.1}$: In this hybrid, Sim'_1 additionally computes each honest server S_j's share of $[\boldsymbol{s}_1]^{(2)}$ by using the input labels associated with the input of S_j to evaluate the circuit $\mathtt{GC}_1^{S_j}$ with Ev. We denote the result of the computation be $\overline{[\boldsymbol{s}_1]^{(2)}}$. Since the input of $\mathtt{Circ}_1^{S_j}$ completely comes from the output of $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$, and the computation process of S_j's share of $[\boldsymbol{s}_1]^{(2)}$ in Π_0 is identical to $\mathtt{Circ}_1^{S_j}$, from the correctness condition of the garbling scheme, the result $\overline{[\boldsymbol{s}_1]^{(2)}}$ is the same as $[\boldsymbol{s}_1]^{(2)}$ from the execution of Π_0. If R_1 is honest, Sim'_1 doesn't generate the honest servers' shares of $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}$ first and then computes each honest server's share of each $[\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}]^{(3)}$ by himself. Instead, Sim'_1 generates $[\boldsymbol{s}_1]^{(2)}$ first and then samples the whole sharing $[\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}]^{(3)}$ based on the corrupted servers' shares of $[\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}]^{(3)}$ and the secret $\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}$. Then, Sim'_1 computes the honest servers' shares of $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}$ based on their shares of $[\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}]^{(3)}, [\boldsymbol{s}_1]^{(2)}, [\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]$. Then for each honest server S_j, Sim'_1 computes each $Y^{S_j}_{a,s_{1,a}^{S_j}}$ by

$$Y^{S_j}_{a,s_{1,a}^{S_j}} = \big(([\boldsymbol{r}_{\boldsymbol{s}_1}^{(1)}]^{(3)})^{S_j}_{[a\ell+1,(a+1)\ell]}, \ldots, ([\boldsymbol{r}_{\boldsymbol{s}_1}^{(\kappa)}]^{(3)})^{S_j}_{[a\ell+1,(a+1)\ell]}\big).$$

The other output labels are still computed by following the protocol. Since the honest servers' shares of each $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}$ are sampled randomly based on corrupted servers' shares and the secret in the last hybrid, their shares of $[\boldsymbol{r}_{\boldsymbol{s}_i}^{(\alpha)}]^{(3)} = [\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)} + [\boldsymbol{s}_1]^{(2)} \otimes [\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]$ are also random based on corrupted servers' shares and the secret. In addition, note that the original computation process of each $Y^{S_j}_{a,s_{1,a}^{S_j}}$ is just the computation of

$$\big(([\boldsymbol{r}_{\boldsymbol{s}_1}^{(1)}]^{(3)})^{S_j}_{[a\ell+1,(a+1)\ell]}, \ldots, ([\boldsymbol{r}_{\boldsymbol{s}_1}^{(\kappa)}]^{(3)})^{S_j}_{[a\ell+1,(a+1)\ell]}\big),$$

we don't change the computation process of $Y^{S_j}_{(a,s_{1,a}^{S_j})}$. Therefore, we only change the order of generating each $[\boldsymbol{r}_{\boldsymbol{s}_1}^{(\alpha)}]^{(3)}$ and $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}$ without changing their distributions. Thus, $\mathbf{Hyb}_{3.1.1}$ and $\mathbf{Hyb}_{3.0}$ have the same distribution.

$\mathbf{Hyb}_{3.1.2}$: In this hybrid, for each honest server S_j, Sim'_1 doesn't follow the protocol to garble $\mathtt{Circ}_1^{S_j}$. Instead, for each wire w in $\mathtt{Circ}_1^{S_j}$ that is not an output wire of an XOR gate or an AND gate, Sim'_1 samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string as $k_{w,v_w\oplus\lambda_w}$. Then Sim'_1 computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$ for these wires. For each wire w in $\mathtt{Circ}^{S_j}$ that is not an output wire of an output gate, Sim'_1 computes $k_{w,1\oplus v_w\oplus\lambda_w} = k_{w,v_w\oplus\lambda_w} \oplus \Delta^{S_j}$. 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, $\mathbf{Hyb}_{3.1.2}$ and $\mathbf{Hyb}_{3.1.1}$ have the same output distribution.

$\mathbf{Hyb}_{3.1.3}$: In this hybrid, Sim_1' maintains a set Q_1. For each honest server S_j, for each AND gate g in $\mathtt{Circ}_1^{S_j}$ with input wire a, b, and for all $i_0, i_1 \in \{0,1\}$ such that $(i_0, i_1) \neq (v_a \oplus \lambda_a, v_b \oplus \lambda_b)$, when the honest server S_j computes $\mathsf{ct}_{i_0,i_1}^{(g)}$, Sim_1' checks whether the query to the random oracle $\mathcal{O}_1$ has been queried before. If true, Sim_1' aborts the simulation. Otherwise, Sim_1' 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 Δ are chosen in the simulation of $\mathsf{Gb}(1^\kappa, \mathtt{Circ}^{S_j}, L)$ (denoted by Δ^{S_j}). Since each query made to the random oracle contains either of the $(\kappa-1)$-bit strings $k_{a,1\oplus v_a\oplus\lambda_a}$ or $k_{b,1\oplus v_b\oplus\lambda_b}$ with $k_{a,1\oplus v_a\oplus\lambda_a} - k_{a,v_a\oplus\lambda_a} = k_{b,1\oplus v_b\oplus\lambda_b} - k_{b,v_b\oplus\lambda_b} = \Delta^{S_j}$ which is uniformly random, for each query made by the adversary, there is only a negligible probability that the query contains $k_{a,1\oplus v_a\oplus\lambda_a}$ or $k_{b,1\oplus v_b\oplus\lambda_b}$. 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 $\mathbf{Hyb}_{3.1.3}$ and $\mathbf{Hyb}_{3.1.2}$ are computationally indistinguishable.

$\mathbf{Hyb}_{3.1.4}$: In this hybrid, for each honest server S_j, for each AND gate g in $\mathtt{Circ}_1^{S_j}$ with input wire a, b and output wire o, and for all $i_0, i_1 \in \{0,1\}$ such that $(i_0, i_1) \neq (v_a \oplus \lambda_a, v_b \oplus \lambda_b)$, Sim_1' samples a random κ-bit string as the ciphertext $\mathsf{ct}_{i_0,i_1}^{(g)}$. While emulating $\mathcal{O}_1$, Sim_1' computes the output based on the random strings and the wire labels of wire o. Note that the only difference between $\mathbf{Hyb}_{3.1.4}$ and $\mathbf{Hyb}_{3.1.3}$ is the way we decide the output for queries in Q_1. Since $\mathsf{ct}_{i_0,i_1}^{(g)}$ is randomly sampled, $\mathsf{ct}_{i_0,i_1}^{(g)} \oplus m$ is also uniformly random for any κ-bit string m. In particular, when Sim_1' does not abort the simulation, queries in Q_1 have not been queried before. Thus, $\mathbf{Hyb}_{3.1.4}$ and $\mathbf{Hyb}_{3.1.3}$ have the same output distribution.

$\mathbf{Hyb}_{3.1.5}$: In this hybrid, for each honest server S_j, Sim_1' change the order of sampling random κ-bit strings as the ciphertext $\mathsf{ct}_{i_0,i_1}^{(g)}$ for each AND gate g in $\mathtt{Circ}_1^{S_j}$ with input wire a, b and output wire o and all $i_0, i_1 \in \{0,1\}$ such that $(i_0, i_1) \neq (v_a \oplus \lambda_a, v_b \oplus \lambda_b)$ and sampling Δ^{S_j} to decide the queries to $\mathcal{O}_1$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{3.1.5}$ and $\mathbf{Hyb}_{3.1.4}$ have the same output distribution.

$\mathbf{Hyb}_{3.1.6}$: In this hybrid, Sim_1' maintains a set Q_2. For each honest server S_j, for each input wire w of an output gate in $\mathtt{Circ}_1^{S_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, when the honest server S_j computes ct_{w,i_2}, Sim_1' checks whether the query to the random oracle $\mathcal{O}_2$ has been queried before. If true, Sim_1' aborts the simulation. Otherwise, Sim_1' adds the query to Q_2. Note that each query made to the random oracle contains a $(\kappa-1)$-bit string $k_{w,1\oplus v_w\oplus\lambda_w}$ with $k_{w,1\oplus v_w\oplus\lambda_w} - k_{w,v_w\oplus\lambda_w} = \Delta^{S_j}$, for the same reason as in $\mathbf{Hyb}_{3.1.3}$, the probability that some query has been queried (either by the honest server or by the adversary) is negligible. Thus, the distributions of $\mathbf{Hyb}_{3.1.6}$ and $\mathbf{Hyb}_{3.1.5}$ are computationally indistinguishable.

$\mathbf{Hyb}_{3.1.7}$: In this hybrid, for each honest server S_j, for each input wire w of an output gate in $\mathtt{Circ}_1^{S_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, Sim_1' samples a random $\ell\kappa$-bit string as the ciphertext ct_{w,i_2}. While emulating $\mathcal{O}_2$, Sim_1' computes the output based on the random strings and the output labels. Note that the only difference between $\mathbf{Hyb}_{3.1.7}$ and $\mathbf{Hyb}_{3.1.6}$ is the way we decide the output for queries in Q_2. Since ct_{w,i_2} is randomly sampled, $\mathsf{ct}_{w,i_2} \oplus m$ is also uniformly random for any $\ell\kappa$-bit string m. In particular, when Sim_2 does not abort the simulation, queries in Q_2 have not been queried before. Thus, $\mathbf{Hyb}_{3.1.7}$ and $\mathbf{Hyb}_{3.1.6}$ have the same output distribution.

Note that for each $\beta = 1, \ldots, k$, if the β-th bit of $\boldsymbol{s}_1$ is 0, then we don't need the key $\boldsymbol{r}_{1,\beta}$ associated with the β-th bit of $\boldsymbol{s}_1$ since it is not used to generate any transcript sent to $\mathcal{A}$ in Step 3-4, Sim_1' delays the generation of $\boldsymbol{r}_{1,\beta}$ when it is needed in the encryption in Step 5 in the next hybrid. Similarly, if the β-th bit of $\boldsymbol{s}_1$ is 1, Sim_1' delays the generation of $\boldsymbol{r}_{0,\beta}$ when it is needed in the encryption in Step 5 in the next hybrid. Then we don't need each sharing $[\boldsymbol{r}_1^{(\alpha)} - \boldsymbol{r}_0^{(\alpha)}]$ in Step 3-4, we also delay the generation of each $[\boldsymbol{r}_1^{(\alpha)} - \boldsymbol{r}_0^{(\alpha)}]$ when it is needed in Step 5.

$\mathbf{Hyb}_{3.1.8}$: In this hybrid, if the receiver R_1 of $[\boldsymbol{s}_1]^{(2)}$ is an honest server S_j and the β-th bit of $\boldsymbol{s}_1$ is used as an input wire with index j_β in $\mathtt{Circ}^{S_j}$, Sim_1' doesn't honestly compute $\mathsf{En}_{j_\beta}(e^{S_j}, s_{1,\beta})$. Instead, Sim_1'

sets the garbled input $X_{j_\beta}^{S_j}$ to be $k_{w_{j_\beta},v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}}}\|(v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}})$. In addition, Sim_1' doesn't honestly compute $\mathsf{En}_\gamma(e^{S_j},x_\gamma)$ for each input wire (with index γ) w_γ of $\mathtt{Circ}^{S_j}$ that does not come from reconstruction. Instead, Sim_1' sets the garbled input $X_\gamma^{S_j}$ to be $k_{w_\gamma,v_{w_\gamma}\oplus\lambda_{w_\gamma}}\|(v_{w_\gamma}\oplus\lambda_{w_\gamma})$ for each w_γ. Since the computation of $\mathsf{En}_{j_\beta}(e^{S_j},s_{1,\beta})$ is just taking $k_{w_{j_\beta},v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}}}\|(v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}})$, and similar for $\mathsf{En}_\gamma(e^{S_j},x_\gamma)$, so this doesn't change the output distribution. Thus, $\mathbf{Hyb}_{3.1.8}$ and $\mathbf{Hyb}_{3.1.7}$ have the same output distribution.

$\mathbf{Hyb}_{3.1.9}$: In this hybrid, if the receiver R_1 of $[\boldsymbol{s}_1]^{(2)}$ is an honest server S_j and the β-th bit of $\boldsymbol{s}_1$ is used as an input wire with index j_β in $\mathtt{Circ}^{S_j}$, when the honest server S_j computes $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1)}$, Sim_1' checks whether the query to the random oracle $\mathcal{O}_1$ has been queried before. If true, Sim_1' aborts the simulation. Otherwise, Sim_1' adds the query to Q_1. Since $\boldsymbol{r}_{1\oplus s_{1,\beta},\beta}$ is generated randomly in $\{0,1\}^\kappa$, the probability that some query has been queried (either by the honest server or by the adversary) is negligible. Thus, the distributions of $\mathbf{Hyb}_{3.1.9}$ and $\mathbf{Hyb}_{3.1.8}$ are computationally indistinguishable.

$\mathbf{Hyb}_{3.1.10}$: In this hybrid, if the receiver R_1 of $[\boldsymbol{s}_1]^{(2)}$ is an honest server S_j and the β-th bit of $\boldsymbol{s}_1$ is used as an input wire with index j_β in $\mathtt{Circ}^{S_j}$, Sim_1' doesn't follow the protocol to compute $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1)}$. Instead, Sim_1' samples a random κ-bit string as $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1)}$. While emulating $\mathcal{O}_1$, Sim_1' computes the output of $\mathcal{O}_1(\boldsymbol{r}_{1\oplus s_{1,\beta},\beta}\|s_{1,\beta}\|1\|\beta\|j_\beta)$ based on $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1)}$ and $k_{w_{j_\beta},1\oplus v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}}}\|(1\oplus v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}})$. Note that the only difference between $\mathbf{Hyb}_{3.1.10}$ and $\mathbf{Hyb}_{3.1.9}$ is the way we decide the output for queries in Q_1. Since $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1)}$ is randomly sampled, $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1)}\oplus m$ is also uniformly random for any κ-bit string m. In particular, when Sim_1' does not abort the simulation, queries in Q_1 have not been queried before. Thus, $\mathbf{Hyb}_{3.1.10}$ and $\mathbf{Hyb}_{3.1.9}$ have the same output distribution.

$\mathbf{Hyb}_{3.1.11}$: In this hybrid, for each honest server S_j, Sim_1' change the order of sampling random κ-bit strings as the ciphertext $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1)}$ and sampling $\boldsymbol{r}_{1\oplus s_{1,\beta},\beta}$ to decide the queries to $\mathcal{O}_1$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{3.1.11}$ and $\mathbf{Hyb}_{3.1.12}$ have the same output distribution.

$\mathbf{Hyb}_{3.1.12}$: In this hybrid, the generation of each sharing $[\boldsymbol{r}_1^{(\alpha)}-\boldsymbol{r}_0^{(\alpha)}]$ is no longer delayed to be done in Step 5. Instead, it is generated at the beginning of the garbling scheme. For each honest server S_j, Sim_1' change the order of sampling random $\ell\kappa$-bit strings as the ciphertext ct_{w,i_2} for each input wire w of an output gate in $\mathtt{Circ}_1^{S_j}$, and for all $i_2\in\{0,1\}$ such that $i_2\neq v_w\oplus\lambda_w$ and sampling Δ^{S_j} to decide the queries to $\mathcal{O}_2$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{3.1.12}$ and $\mathbf{Hyb}_{3.1.11}$ have the same output distribution.

Note that for each honest server S_j, Δ^{S_j} is not used before $\mathtt{GC}_1^{S_j}$ is generated. Sim_1' delays the generating of Δ^{S_j} after $\mathtt{GC}_1^{S_j}$ is generated.

Similarly, for each $\gamma=2,\ldots,\mathsf{rec}$ we can define $\mathbf{Hyb}_{3.\gamma.1},\ldots,\mathbf{Hyb}_{3.\gamma.12}$.

$\mathbf{Hyb}_{3.\gamma.1}$: In this hybrid, Sim_1' additionally computes each honest server S_j's share of $[\boldsymbol{s}_\gamma]^{(2)}$ by using the input labels associated with the input of S_j to evaluate the circuit $\mathtt{GC}_\gamma^{S_j}$ with Ev. We denote the result of the computation be $\overline{[\boldsymbol{s}_\gamma]^{(2)}}$. Since the input of $\mathtt{Circ}_2^{S_j}$ completely comes from the output of $\mathcal{F}_{\mathsf{prep}},\mathcal{F}_{\mathsf{input}}$ 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 $\mathtt{Circ}_\gamma^{S_j}$, from the correctness condition of the garbling scheme, the result $\overline{[\boldsymbol{s}_\gamma]^{(2)}}$ is the same as $[\boldsymbol{s}_\gamma]^{(2)}$ from the execution of Π_0. If R_1 is honest, Sim_1' doesn't generate the honest servers' shares of $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}$ first and then computes each honest server's shares of each $[\boldsymbol{r}_{\boldsymbol{s}_\gamma}^{(\alpha)}]^{(3)}$ by himself. Instead, Sim_1' generates $[\boldsymbol{s}_\gamma]^{(2)}$ first and then samples the whole sharing $[\boldsymbol{r}_{\boldsymbol{s}_\gamma}^{(\alpha)}]^{(3)}$ based on the corrupted servers' shares of $[\boldsymbol{r}_{\boldsymbol{s}_\gamma}^{(\alpha)}]^{(3)}$ and the secret. Then, Sim_1' computes the honest servers' shares of $[\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]^{(3)}$ based on their shares of $[\boldsymbol{r}_{\boldsymbol{s}_\gamma}^{(\alpha)}]^{(3)},[\boldsymbol{s}_\gamma]^{(2)},[\boldsymbol{r}_{\boldsymbol{1}}^{(\alpha)}-\boldsymbol{r}_{\boldsymbol{0}}^{(\alpha)}]$. Then for each honest server S_j, Sim_1' computes each $Y_{(\gamma-1)\ell^2+a,s_{\gamma,a}^{S_j}}^{S_j}$ by

$$Y_{(\gamma-1)\ell^2+a,s_{\gamma,a}^{S_j}}^{S_j}=\big(([\boldsymbol{r}_{\boldsymbol{s}_\gamma}^{(1)}]^{(3)})_{[a\ell+1,(a+1)\ell]}^{S_j},\ldots,([\boldsymbol{r}_{\boldsymbol{s}_\gamma}^{(\kappa)}]^{(3)})_{[a\ell+1,(a+1)\ell]}^{S_j}\big).$$

The other output labels are still computed by following the protocol. For the same reason as in $\mathbf{Hyb}_{3.1.1}$, we conclude that $\mathbf{Hyb}_{3.\gamma.1}$ and $\mathbf{Hyb}_{3.(\gamma-1).8}$ have the same output distribution.

$\mathbf{Hyb}_{3.\gamma.2}$: In this hybrid, for each honest server S_j, Sim'_1 doesn't follow the protocol to garble $\mathtt{Circ}_\gamma^{S_j}$. Instead, for each wire w in $\mathtt{Circ}_\gamma^{S_j}$ that is not an output wire of an XOR gate or an AND gate, Sim'_1 samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string as $k_{w,v_w\oplus\lambda_w}$. Then Sim'_1 computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$ for these wires. For each wire w in $\mathtt{Circ}^{S_j}$ that is not an output wire of an output gate, Sim'_1 computes $k_{w,1\oplus v_w\oplus\lambda_w} = k_{w,v_w\oplus\lambda_w} \oplus \Delta^{S_j}$. For the same reason in $\mathbf{Hyb}_{3.1.2}$, $\mathbf{Hyb}_{3.\gamma.2}$ and $\mathbf{Hyb}_{3.\gamma.1}$ have the same output distribution.

$\mathbf{Hyb}_{3.\gamma.3}$: In this hybrid, for each honest server S_j, for each AND gate g in $\mathtt{Circ}_\gamma^{S_j}$ with input wire a, b, and for all $i_0, i_1 \in \{0,1\}$ such that $(i_0, i_1) \neq (v_a \oplus \lambda_a, v_b \oplus \lambda_b)$, when the honest server S_j computes $\mathsf{ct}_{i_0,i_1}^{(g)}$, Sim'_1 checks whether the query to the random oracle $\mathcal{O}_1$ has been queried before. If true, Sim'_1 aborts the simulation. Otherwise, Sim'_1 adds the query to Q_1. For the same reason in $\mathbf{Hyb}_{3.1.3}$, the distributions of $\mathbf{Hyb}_{3.\gamma.3}$ and $\mathbf{Hyb}_{3.\gamma.2}$ are computationally indistinguishable.

$\mathbf{Hyb}_{3.\gamma.4}$: In this hybrid, for each honest server S_j, for each AND gate g in $\mathtt{Circ}_\gamma^{S_j}$ with input wire a, b and output wire o, and for all $i_0, i_1 \in \{0,1\}$ such that $(i_0, i_1) \neq (v_a \oplus \lambda_a, v_b \oplus \lambda_b)$, Sim'_1 samples a random κ-bit string as the ciphertext $\mathsf{ct}_{i_0,i_1}^{(g)}$. While emulating $\mathcal{O}_1$, Sim'_1 computes the output based on the random strings and the wire labels of wire o. For the same reason in $\mathbf{Hyb}_{3.1.4}$, $\mathbf{Hyb}_{3.\gamma.4}$ and $\mathbf{Hyb}_{3.\gamma.3}$ have the same output distribution.

$\mathbf{Hyb}_{3.\gamma.5}$: In this hybrid, for each honest server S_j, Sim'_1 change the order of sampling random κ-bit strings as the ciphertext $\mathsf{ct}_{i_0,i_1}^{(g)}$ for each AND gate g in $\mathtt{Circ}_\gamma^{S_j}$ with input wire a, b and output wire o and all $i_0, i_1 \in \{0,1\}$ such that $(i_0, i_1) \neq (v_a \oplus \lambda_a, v_b \oplus \lambda_b)$ and sampling Δ^{S_j} to decide the queries to $\mathcal{O}_1$. For the same reason in $\mathbf{Hyb}_{3.1.5}$, $\mathbf{Hyb}_{3.\gamma.5}$ and $\mathbf{Hyb}_{3.\gamma.4}$ have the same output distribution.

$\mathbf{Hyb}_{3.\gamma.6}$: In this hybrid, for each honest server S_j, for each input wire w of an output gate in $\mathtt{Circ}_\gamma^{S_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, when the honest server S_j computes ct_{w,i_2}, Sim'_1 checks whether the query to the random oracle $\mathcal{O}_2$ has been queried before. If true, Sim'_1 aborts the simulation. Otherwise, Sim'_1 adds the query to Q_2. For the same reason in $\mathbf{Hyb}_{3.1.6}$, the distributions of $\mathbf{Hyb}_{3.\gamma.6}$ and $\mathbf{Hyb}_{3.\gamma.5}$ are computationally indistinguishable.

$\mathbf{Hyb}_{3.\gamma.7}$: In this hybrid, for each honest server S_j, for each input wire w of an output gate in $\mathtt{Circ}_\gamma^{S_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, Sim'_1 samples a random $\ell\kappa$-bit string as the ciphertext ct_{w,i_2}. While emulating $\mathcal{O}_2$, Sim'_1 computes the output based on the random strings and the output labels. For the same reason in $\mathbf{Hyb}_{3.1.7}$, $\mathbf{Hyb}_{3.\gamma.7}$ and $\mathbf{Hyb}_{3.\gamma.6}$ have the same output distribution.

$\mathbf{Hyb}_{3.\gamma.8}$: In this hybrid, if the receiver R_γ of $[\boldsymbol{s}_\gamma]^{(2)}$ is an honest server S_j and the β-th bit of $\boldsymbol{s}_\gamma$ is used as an input wire with index j_β in $\mathtt{Circ}^{S_j}$, Sim'_1 doesn't honestly compute $\mathsf{En}_{j_\beta}(e^{S_j}, s_{\gamma,\beta})$. Instead, Sim'_1 sets the garbled input $X_{j_\beta}^{S_j}$ to be $k_{w_{j_\beta},v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}}} \| (v_{w_{j_\beta}} \oplus \lambda_{w_{j_\beta}})$. In addition, Sim'_1 doesn't honestly compute $\mathsf{En}_\eta(e^{S_j}, x_\eta)$ for each input wire (with index η) w_η of $\mathtt{Circ}^{S_j}$ that does not come from reconstruction. Instead, Sim'_1 sets the garbled input $X_\eta^{S_j}$ to be $k_{w_\eta,v_{w_\eta}\oplus\lambda_{w_\eta}} \| (v_{w_\eta} \oplus \lambda_{w_\eta})$ for each w_η. For the same reason in $\mathbf{Hyb}_{3.1.8}$, $\mathbf{Hyb}_{3.\gamma.8}$ and $\mathbf{Hyb}_{3.\gamma.7}$ have the same output distribution.

$\mathbf{Hyb}_{3.\gamma.9}$: In this hybrid, if the receiver R_γ of $[\boldsymbol{s}_\gamma]^{(2)}$ is an honest server S_j and the β-th bit of $\boldsymbol{s}_\gamma$ is used as an input wire with index j_β in $\mathtt{Circ}^{S_j}$, when the honest server S_j computes $\mathsf{ct}_{j_\beta,1\oplus s_{\gamma,\beta}}^{(1)}$, Sim'_1 checks whether the query to the random oracle $\mathcal{O}_1$ has been queried before. If true, Sim'_1 aborts the simulation. Otherwise, Sim'_1 adds the query to Q_1. For the same reason in $\mathbf{Hyb}_{3.1.9}$, the distributions of $\mathbf{Hyb}_{3.\gamma.9}$ and $\mathbf{Hyb}_{3.\gamma.8}$ are computationally indistinguishable.

$\mathbf{Hyb}_{3.\gamma.10}$: In this hybrid, if the receiver R_γ of $[\boldsymbol{s}_\gamma]^{(2)}$ is an honest server S_j and the β-th bit of $\boldsymbol{s}_\gamma$ is used as an input wire with index j_β in $\mathtt{Circ}^{S_j}$, Sim'_1 doesn't follow the protocol to compute $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}$. Instead, Sim'_1 samples a random κ-bit string as $\mathsf{ct}_{j_\beta,1\oplus s_{\gamma,\beta}}^{(\gamma)}$. While emulating $\mathcal{O}_1$, Sim'_1 computes the output of $\mathcal{O}_1(\boldsymbol{r}_{1\oplus s_{\gamma,\beta},\beta})$ based on $\mathsf{ct}_{j_\beta,1\oplus s_{\gamma,\beta}}$ and $k_{w_{j_\beta},1\oplus v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}}} \| (1 \oplus v_{w_{j_\beta}} \oplus \lambda_{w_{j_\beta}})$. For the same reason in $\mathbf{Hyb}_{3.1.10}$,

$\mathbf{Hyb}_{3.\gamma.10}$ and $\mathbf{Hyb}_{3.\gamma.9}$ have the same output distribution.

$\mathbf{Hyb}_{3.\gamma.11}$: In this hybrid, for each honest server S_j, Sim_1' change the order of sampling random κ-bit strings as the ciphertext $\mathsf{ct}^{(\gamma)}_{j_\beta,1\oplus s_{\gamma,\beta}}$ and sampling $\boldsymbol{r}_{1\oplus s_{\gamma,\beta},\beta}$ to decide the queries to $\mathcal{O}_1$. For the same reason in $\mathbf{Hyb}_{3.1.11}$, $\mathbf{Hyb}_{3.\gamma.11}$ and $\mathbf{Hyb}_{3.\gamma.12}$ have the same output distribution.

$\mathbf{Hyb}_{3.\gamma.12}$: In this hybrid, the generation of each sharing $[\boldsymbol{r}_1^{(\alpha)} - \boldsymbol{r}_0^{(\alpha)}]$ is no longer delayed to be done in Step 5. Instead, it is generated at the beginning of the garbling scheme. For each honest server S_j, Sim_1' change the order of sampling random $\ell\kappa$-bit strings as the ciphertext ct_{w,i_2} for each input wire w of an output gate in $\mathtt{Circ}_\gamma^{S_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$ and sampling Δ^{S_j} to decide the queries to $\mathcal{O}_2$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{3.\gamma.12}$ and $\mathbf{Hyb}_{3.\gamma.11}$ have the same output distribution.

Note that for each honest server S_j, Δ^{S_j} is not used before $\mathtt{GC}_\gamma^{S_j}$ is generated. Sim_1' delays the generating of Δ^{S_j} after $\mathtt{GC}_\gamma^{S_j}$ is generated.

Note that $\mathbf{Hyb}_{3.\mathsf{rec}.12}$ is just $\mathbf{Hyb}_3$, we conclude that $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same distribution. For each honest server S_j, Sim_1' has delayed the generating of Δ^{S_j} after the whole garbled circuit $\mathtt{GC}^{S_j}$ is generated.

Also note that if R_i is an honest client, $\boldsymbol{r}^{(\alpha)}_{\mathbf{1}\oplus \boldsymbol{s}_i}$ for $\alpha = 1, \ldots, \kappa$ are not used until the whole garbled circuit $\mathtt{GC}^{S_j}$ is generated. Sim_1' generates them after the whole garbled circuit is generated in future hybrids to decide the set Q_1.

$\mathbf{Hyb}_4$: In this hybrid, for each AND gate g in each honest server S_j's local circuit $\mathtt{Circ}^{S_j}$ with input wire a, b and output wire o, Sim_1' doesn't compute the output of $\mathcal{O}_1$ to the queries that are used to generate these ciphertexts based on the random strings and the wire labels of wire o. For each output gate in $\mathtt{Circ}^{S_j}$ indexed $1, \ldots, f.m$ with input wire w, Sim_1' doesn't compute the output of $\mathcal{O}_2$ 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 query to the random oracle when S_j is computing the cipher-texts has been queried before. We prove that the distributions of $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ are computationally indistinguishable.

For the sake of contradiction, assume that there exists an adversary $\mathcal{A}_1$ such that $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ are computationally distinguishable. Let Q_1, Q_2 be the set of queries to the random oracles $\mathcal{O}_1, \mathcal{O}_2$ respectively when S_j computes his ciphertexts 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 $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_4$ 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 $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ 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 $\mathbf{Hyb}_4$. However, each query in Q_1, Q_2 either contains $k_{w,1\oplus v_w\oplus\lambda_w}$ for a wire w in some honest server's circuit or contains $\boldsymbol{r}_{1\oplus s_{i,\beta},\beta}$ for some $\beta \in \{1, \ldots, k\}$. Suppose there is one query that contains $k_{w,1\oplus v_w\oplus\lambda_w}$ for a wire w in an honest server S_j's circuit $\mathtt{Circ}^{S_j}$. Since Δ^{S_j} 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 Δ^{S_j}. Therefore, $k_{w,1\oplus v_w\oplus\lambda_w} = k_{w,v_w\oplus\lambda_w} \oplus \Delta^{S_j}$ only has $2^{-\kappa+1} \cdot \mathsf{poly}(\kappa)$ probability to be queried by $\mathcal{A}$, which is negligible. Similarly, if one query contains $\boldsymbol{r}_{1\oplus s_{i,\beta},\beta}$, since $\boldsymbol{r}_{1\oplus s_{i,\beta},\beta}$ is not used in computing any message before all the ciphertexts are generated, we can generate it after the garbled circuit is generated, and the probability that it is contained in a query is negligible. Thus, the distributions of $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ are computationally indistinguishable.

Note that for honest server S_j, Δ^{S_j} is not used in the simulation, Sim_1' doesn't generate it in future hybrids.

Also note that if R_i is an honest client, $\boldsymbol{r}^{(\alpha)}_{\mathbf{1}\oplus \boldsymbol{s}_i}$ for $\alpha = 1, \ldots, \kappa$ are not used until R_i receives $\boldsymbol{s}_i, \{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}\}_{\alpha=1}^{\kappa}$ and does the verification on them. Sim_1' delays the generation of $\{\boldsymbol{r}^{(\alpha)}_{\mathbf{1}\oplus \boldsymbol{s}_i}\}_{\alpha=1}^{\kappa}$ after R_i receives $\boldsymbol{s}_i, \{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}\}_{\alpha=1}^{\kappa}$

in future hybrids.

Hyb$_5$: In this hybrid, for each $i = 1, \ldots, \mathsf{rec}$ with honest receiver R_i who is a client, Sim'_1 doesn't follow the protocol to check the values $\boldsymbol{s}_i, \{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}\}^{\kappa}_{\alpha=1}$ received from P_{king}. Instead, Sim'_1 checks whether they are correctly sent. Note that when they are correctly sent, then $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i} = \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}} + \boldsymbol{s}_i * (\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}})$ must hold for each $\alpha = 1, \ldots, \kappa$. Thus, the output only changes when $\boldsymbol{s}_i, \{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}\}^{\kappa}_{\alpha=1}$ are not correctly sent but for each $\alpha = 1, \ldots, \kappa$ it still holds that $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i} = \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}} + \boldsymbol{s}_i * (\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}})$ (by the values received from P_{king}). Since when $\boldsymbol{s}_i$ is correctly sent, $\{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}\}^{\kappa}_{\alpha=1}$ is determined by the equation $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i} = \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}} + \boldsymbol{s}_i * (\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}})$, the output only changes when a bit of $\boldsymbol{s}_i$ is not correctly sent. Assume that it's the β-th bit. Note that if the β-th bit of $\boldsymbol{s}_i$ is 0, then the β-th bit of $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}$ is the β-th bit of $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}}$, which is sampled randomly after $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}, \boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}}$ are received from P_{king}. Similarly, if the β-th bit of $\boldsymbol{s}_i$ is 1, then the β-th bit of $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}$ is the β-th bit of $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}$, which is also sampled randomly. Thus, the output changes only when κ randomly sampled bits are all guessed correctly by $\mathcal{A}$. The probability is $2^{-\kappa}$, which is negligible. Thus, the distributions of **Hyb**$_5$ and **Hyb**$_4$ are statistically close.

Note that if R_i is an honest client, we only need $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i}$ and we don't need $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}, \boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}}$ for the simulation, and we also don't need honest servers' shares of $\{[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1}} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0}}]\}^{\kappa}_{\alpha=1}$ in the simulation. Sim'_1 doesn't generate them in future hybrids.

Hyb$_6$: In this hybrid, since all the transcripts between honest and corrupted parties generated by Sim'_1 can be generated from the transcripts between honest and corrupted parties obtained in the execution of Π_0, Sim'_1 just runs Π_0 first to obtain all the transcripts and then uses them to generate the output of Sim'_1. In addition, honest clients don't follow the protocol Π'_1 to compute their output. Instead, they follow Π_0 to get their output. Since the value $\boldsymbol{s}_i$ sent from P_{king} to each honest client in Π'_1 is the same as the value $\boldsymbol{s}_i$ in Π_0, and the preprocessing and input data of Π_0, Π'_1 is also the same, the computation of honest clients' output is in the two protocols is completely the same. Therefore, we only change the way of generating the output of Sim'_1 without changing their distributions. Thus, **Hyb**$_6$ and **Hyb**$_5$ have the same distribution.

Hyb$_7$: In this hybrid, Sim'_1 doesn't run Π_0 to get the transcripts between honest and corrupted parties in Π_0 and use them to all the transcripts between honest and corrupted parties in Π'_1. Instead, Sim'_1 invokes Sim_0 with $\mathcal{A}'$ to get the transcripts between honest and corrupted parties in Π_0. In addition, honest clients get their outputs from $\mathcal{F}$ instead of following Π_0 to compute them. From the requirements of Π_0, the joint distribution of transcripts between honest and corrupted parties in Π_0 and the honest clients' output in Π_0 is computationally indistinguishable from the joint distribution of the output of Sim_0 and honest clients' output from $\mathcal{F}$. Thus, the distributions of **Hyb**$_7$ and **Hyb**$_6$ are computationally indistinguishable.

Note that **Hyb**$_7$ is the ideal-world scenario, Π'_1 computes $\mathcal{F}$ with computational security.

□

E Cost Analysis for Π'_1

In this section, we give a detailed cost analysis for protocol Π'_1 step by step as follows:

1. **Preprocessing.** This step only contains an invocation of $\mathcal{F}_{\mathsf{prep}}$. Suppose that $\mathcal{F}_{\mathsf{prep}}$ is realized with communication cost $\mathsf{CC}_{\mathsf{prep}}$, then the communication cost of this step is $\mathsf{CC}_{\mathsf{prep}}$.

2. **Input.** This step only contains an invocation of $\mathcal{F}_{\mathsf{input}}$. Suppose that $\mathcal{F}_{\mathsf{input}}$ is realized with communication cost $\mathsf{CC}_{\mathsf{input}}$, then the communication cost of this step is $\mathsf{CC}_{\mathsf{input}}$.

3. **Generating Output Labels.** In this step, for each $i = 1, \ldots, \mathsf{rec}$, R_i needs to distribute κ $\Sigma^{(3)}$-sharings and κ Σ-sharings to all the parties. Note that the communication of the evaluation phase of Π_0 comes from rec reconstructions of $\Sigma^{(2)}$-sharings, each of size $n\ell^2$, while the size of κ $\Sigma^{(3)}$-sharings and κ Σ-sharings is $O(n\ell^3\kappa)$. Thus, the communication cost of this step is $O(\ell\kappa \cdot \mathsf{CC}^{\Pi_0}_{\mathsf{eval}})$, where $\mathsf{CC}^{\Pi_0}_{\mathsf{eval}}$ is the communication cost of the evaluation phase of Π_0.

4. **Garbling Local Circuits.** In this step, the communication comes from sending each S_j's $(\mathtt{GC}_1^{S_j}, \ldots, \mathtt{GC}_{\mathsf{rec}}^{S_j}, d^{S_j})$ to P_{king}. Let

$$L^{S_j} = ((Y_{1,0}^{S_j}, Y_{1,1}^{S_j}), \ldots, (Y_{\ell^2\mathsf{rec},0}^{S_j}, Y_{\ell^2\mathsf{rec},1}^{S_j})),$$

the size of each server S_j's garbled circuits is $O((G_I^{S_j} + G_A^{S_j}) \cdot \kappa + |L^{S_j}|)$, where $G_I^{S_j}, G_A^{S_j}$ are the number of distinct input wires and AND gates of S_j's circuits respectively. Note that the sum of the number of all the servers' distinct input wires of their local circuits is exactly the size of the preprocessing and input data, i.e. the output size of $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$, we denote the total output size of the two functionalities by DS. In addition, $L^{S_1}, \ldots, L^{S_n}$ contains two labels of size $\ell\kappa$ for each bit communicated in the evaluation phase of Π_0. Thus, the size of $(L^{S_1}, \ldots, L^{S_n})$ is $\sum_{j=1}^{n} |L^{S_j}| = O(\ell\kappa \cdot \mathsf{CC}_{\mathsf{eval}}^{\Pi_0})$. In addition, d^{S_j} contains a masking bit for each output wire of S_j's local circuit which corresponds to a bit communicated in the evaluation phase of Π_0, so the sum of d^{S_j} for all the server is just $\mathsf{CC}_{\mathsf{eval}}^{\Pi_0}$. To sum up, the total communication cost of this step is $O((\mathsf{DS} + G_A) \cdot \kappa + \ell\kappa \cdot \mathsf{CC}_{\mathsf{eval}}^{\Pi_0})$, where G_A is the total number of AND gates in the servers' local circuits.

5. **Encrypting Input Labels.** In this step, the servers need to send two ciphertexts of size $O(\kappa)$ for each bit communicated from one server to another in the evaluation phase of Π_0, so the communication cost of this step is $O(\kappa \cdot \mathsf{CC}_{\mathsf{eval}}^{\Pi_0})$.

6. **Sending Input Labels.** In this step, each server sends the labels of the preprocessing and input data received from $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ to P_{king}, where each label is of size κ. Thus, the total cost is bounded by $O(\mathsf{DS} \cdot \kappa)$.

7. **Evaluating the Circuit.** This step only contains local computation.

8. **Sending Outputs.** In this step, the communication comes from κ secrets of $\Sigma^{(3)}$-sharings and a secret of a $\Sigma^{(2)}$-sharing for each reconstruction of a $\Sigma^{(2)}$-sharing with client receiver in the evaluation phase of Π_0. The size of κ secrets of $\Sigma^{(3)}$-sharings and a secret of a $\Sigma^{(2)}$-sharing is $O(k\kappa)$ while the size of a $\Sigma^{(2)}$-sharing is $O(n\ell^2)$. Thus, the communication cost of this step is $O(k\kappa\mathsf{CC}_{\mathsf{eval}}^{\Pi_0}/n\ell^2)$. Since the secret size of an LSSS can't be larger than the size of the whole sharing, we have $k \leq n\ell$, so the communication cost of this step is $O(\kappa\mathsf{CC}_{\mathsf{eval}}^{\Pi_0}/\ell)$.

As analyzed above, the total communication cost of Π_1' is

$$\mathsf{CC}^{\Pi_1'} = \mathsf{CC}_{\mathsf{prep}} + \mathsf{CC}_{\mathsf{input}} + O((\mathsf{DS} + G_A) \cdot \kappa + \ell\kappa \cdot \mathsf{CC}_{\mathsf{eval}}^{\Pi_0}).$$

F Instantiation of the Abstract Protocol

In Section 6, we have given an outline of instantiating the abstract protocol for SIMD circuits. In this section, we show how to instantiate the abstract protocol for general circuits.

F.1 Handling Network Routing.

Recall in Section 6, we have reduced the task of evaluating a general circuit to the following two steps:

- For each group of addition/multiplication gates, given the two input sharings $[\boldsymbol{x}], [\boldsymbol{y}]$, compute the output sharing $[\boldsymbol{z}]$, where $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$ for addition gates and $\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{y}$ for multiplication gates.
- Given output sharings from all previous layers, prepare the input sharings for the current layer.

The second step is to prepare the input sharings of each layer by using the output sharings from previous layers. While this holds automatically for SIMD circuits, for general circuits, the input sharings do not come for free due to the following issues:

- The secrets needed to be in a single sharing may be scattered in different output sharings of previous layers.
- Even if we have all the secrets in a single sharing, we need the secrets to be in the correct order so that the i-th secret is the input of the i-th gate.

This problem is referred to as *network routing* [GPS21, GPS22].

In [GPS21, GPS22], the authors reduce the problem of network routing to the sharing transformation problem. A sharing transformation problem is to transform an (n, t, k, ℓ)-LSSS sharing $[\boldsymbol{s}]$ to $[L(\boldsymbol{s})]$, where $L : \mathbb{F}_2^k \to \mathbb{F}_2^k$ is an $\mathbb{F}$-linear map. With a sharing transformation scheme, after we obtain output Σ-sharings for each group of gates in the current layer, the authors in [GPS21] show that the input Σ-sharings in the next layer can be obtained as follows:

- For each Σ-sharing in the current layer, we perform the fan-out operation to copy each secret enough number of times. For example, if a Σ-sharing $[x_1, x_2, x_3]$ satisfies that x_1, x_2, x_3 will be used by $2, 3, 1$ times in future layers respectively, then all parties compute $[x_1, x_1, x_2]$ and $[x_2, x_2, x_3]$. Note that all the secrets of desired sharings can be obtained by applying linear transformations on the secret of the original sharing. Thus, the desired sharings can be obtained via sharing transformation.
- After doing the fan-out operations, for each obtained Σ-sharing, we perform a proper permutation on the secrets, which is also a linear transformation. This can also be done by sharing transformation.
- After completing the above two steps, we move to prepare the input Σ-Sharings we need in the next layer. The main property that is achieved in [GPS21] is that, for every Σ-sharing $[\boldsymbol{x}]$ we want to prepare, the previous steps have generated k Σ-sharings $[\boldsymbol{x}^{(1)}], \ldots, [\boldsymbol{x}^{(k)}]$ such that there exists a permutation $p : \{1, \ldots, k\} \to \{1, \ldots, k\}$ and $x_i = x^{(i)}_{p(i)}$, where x_i and $x^{(i)}_{p(i)}$ denotes the the i-th bit of $\boldsymbol{x}$ and the $p(i)$-th bit of $\boldsymbol{x}^{(i)}$ respectively. In other words, we only need to deal with a much simpler task of collecting secrets from different positions. The obtained sharings is of form $[\boldsymbol{x}']$ with the $p(i)$-th bit of $\boldsymbol{x}'$ is equal to $x^{(i)}_{p(i)} = x_i$. For this secret collection task, we will discuss it in Section F.3.2.
- Finally, to obtain $[\boldsymbol{x}]$, we permute the secrets in $[\boldsymbol{x}']$, which again is a linear transformation. Thus, this can also be done via sharing transformation.

Our instantiation uses [GPS21] in a black-box way to find the above linear maps, and we refer the readers to [GPS21] for more details about how to find the linear maps.

More concretely, the clients and servers should first perform a circuit transformation to turn a general circuit into a circuit whose gates of each layer can be batched into groups of k gates in the same type. A difference between our instantiation and the protocol [GPS21] is that ours additionally requires that the servers can only do reconstructions of $\Sigma^{(2)}$-sharings in the circuit evaluation phase of our protocol apart from local computations. To achieve this requirement, we provide a protocol $\Pi_{\mathsf{Transpose}}$ and use it to evaluate each k groups of k multiplication gates together (k^2 gates in total) and perform k sharing transformations together. (As a result, we need each layer to have at least k^2 gates.)

Then, for each group of gates, we add fan-out gates that copy the output wires enough times to make sure that each output wire of the input layer and all intermediate layers is used exactly once as an input wire of a later layer. Then the input and output wires in each layer can be written as two $N \times k$ matrices $\mathcal{I}, \mathcal{O}$ with $N = O(|C|/n)$. The only thing we need to do is to permute the entries of $\mathcal{O}$ to the entries of $\mathcal{I}$ to let the output Σ-sharings of former layers be used as input sharings for later layers. To this end, the following lemma has been shown by Hall's Marriage Theorem:

Lemma 3. ([GPS21]) *Let $\ell \geq 1, k \geq 1$ be integers. Let $\mathcal{N}$ be a matrix of dimension $\ell \times k$ in $\{1, 2, \ldots, \ell\}^{\ell \times k}$ such that for all $i \in \{1, \ldots, \ell\}$, the number of entries of $\mathcal{N}$ which are equal to i is k. Then, there exists ℓ permutations $p_1, \ldots, p_\ell$ over $\{1, 2, \ldots, k\}$ such that after performing the permutation p_i on the i-th row of $\mathcal{N}$, the new matrix $\mathcal{N}'$ satisfies that each column of $\mathcal{N}'$ is a permutation over $\{1, 2, \ldots, \ell\}$ the permutations $p_1, \ldots, p_\ell$ can be found within polynomial time.*

As a result of the above lemma, the Σ-sharing of each batch of input wire in a layer can be obtained by performing sharing transformations [GPS22] and secret collections from different positions on the Σ-sharings of output wires from former layers. As a result, we only need to handle batched addition/multiplication gates, sharing transformations, and secret collections from different positions to evaluate the whole circuit. We have shown how to handle batched addition/multiplication gates in Section 6, and we will show how to deal with the remaining tasks in Section F.3.2.

F.2 Instantiation of Linear Secret Sharing Scheme.

We now begin to present the formal description of our protocol Π_0. We first instantiate the LSSSs Σ together with $\Sigma^{(2)}, \Sigma^{(3)}$. Our secret sharing scheme Σ is based on the secret sharing scheme Σ' given in [CC06] based on algebraic geometry. For additional preliminaries about algebraic geometry, we refer the readers to Section B. In short, n parties 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 present a lemma proved in [CC06].

Lemma 4. ([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 $\mathsf{deg}(E) + 1$ $\mathbb{F}_q$-rational points on C outside the support of E.*

Lemma 4 implies that the secret $\boldsymbol{s}$ of $[\boldsymbol{s}]$ can be reconstructed from any $\mathsf{deg}(D) + 1$ parties' shares, which implies the reconstruction algorithm $\Sigma'.\mathsf{Rec}$. In [CC06], the authors have shown that Σ' is an $(n, t, k, 1)$-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 $\mathbb{F}_2$. Besides, each party's share in $\mathbb{F}_q = \mathbb{F}_{2^\ell}$ can be written as a vector in over $\mathbb{F}_2^\ell$. In this way, we obtain an (n, t, k, ℓ)-LSSS over $\mathbb{F}_2$. We take such a sharing to be Σ.

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 $(\mathsf{Conv}^{-1}(y_1), \ldots, \mathsf{Conv}^{-1}(y_n))$. We define Σ 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 $\Sigma^{(2)}$. Now we begin to define $\Sigma^{(2)}$. We first define an LSSS $\Sigma^{(2)'}$ over $\mathbb{F}_q$, denoted by $[\cdot]^{(2)'}$. $\Sigma^{(2)'}$ is the same as Σ' except that the divisor D is replaced by $2D$ (i.e. f is selected from $\mathcal{L}(2D)$). Note that for any $f_1, f_2 \in \mathcal{L}(D)$, it holds that:

$$(\mathsf{div}(f_1) + D) + (\mathsf{div}(f_2) + D) = \mathsf{div}(f_1 \cdot f_2) + 2D,$$

which implies that $f_1 \cdot f_2 \in \mathcal{L}(2D)$. Thus, the multiplication of two sharings $[\boldsymbol{s}_1]' \cdot [\boldsymbol{s}_2]'$ is a $\Sigma^{(2)'}$-sharing $[\boldsymbol{s}_1 * \boldsymbol{s}_2]^{(2)'}$. Note that for any $x^{(1)}, x^{(2)} \in \mathbb{F}_q$ with $\mathsf{Conv}^{-1}(x^{(1)}) = (x_1^{(1)}, \ldots, x_\ell^{(1)}), \mathsf{Conv}^{-1}(x^{(2)}) = (x_1^{(2)}, \ldots, x_\ell^{(2)}) \in \mathbb{F}_2^\ell$, their product $x^{(1)} \cdot x^{(2)}$ is uniquely determined by $(x_1^{(1)}, \ldots, x_\ell^{(1)}) \otimes (x_1^{(2)}, \ldots, x_\ell^{(2)}) \in \mathbb{F}_2^{\ell^2}$. 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 $\mathsf{Conv}_2(\boldsymbol{y}) = x$.

We define $\Sigma^{(2)}$ as 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 $(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n)$,

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

Defining $\Sigma^{(3)}$. Similarly, we may define $\Sigma^{(3)'}$ over $\mathbb{F}_q$ by replacing D by $3D$ in Σ'. For any $x^{(1)}, x^{(2)}, x^{(3)} \in \mathbb{F}_q$ with $\mathsf{Conv}^{-1}(x^{(1)}) = (x_1^{(1)}, \ldots, x_\ell^{(1)}), \mathsf{Conv}^{-1}(x^{(2)}) = (x_1^{(2)}, \ldots, x_\ell^{(2)}) \in \mathbb{F}_2^\ell, \mathsf{Conv}^{-1}(x^{(3)}) = (x_1^{(3)}, \ldots, x_\ell^{(3)}) \in \mathbb{F}_2^\ell$, their product $x^{(1)} \cdot x^{(2)} \cdot x^{(3)}$ is uniquely determined by $(x_1^{(1)}, \ldots, x_\ell^{(1)}) \otimes (x_1^{(2)}, \ldots, x_\ell^{(2)}) \otimes (x_1^{(3)}, \ldots, x_\ell^{(3)}) \in \mathbb{F}_2^{\ell^3}$. 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 $\mathsf{Conv}_3(\boldsymbol{y}) = x$.

We define $\Sigma^{(3)}$ 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 $(\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n)$,

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

Parameter Choices. Let q be a square, there exists a family of curves [GS96] $\{C^{(j)}\}_{j=1}^\infty$ such that

$$\#C^{(j)}(\mathbb{F}_q) \geq (q - \sqrt{q}) \cdot \sqrt{q}^{j-1} \quad \text{and} \quad g(C^{(j)}) \leq \sqrt{q}^j.$$

Lemma 4 implies that the secret $\boldsymbol{s}$ of $[\boldsymbol{s}]$ can be reconstructed from any $\mathsf{deg}(D) + 1$ parties' shares. Since for any $f_1, f_2 \in \mathcal{L}(D)$, it holds that:

$$(\mathsf{div}(f_1) + D) + (\mathsf{div}(f_2) + D) = \mathsf{div}(f_1 \cdot f_2) + 2D.$$

Hence $f_1 \cdot f_2 \in \mathcal{L}(2D)$, so we need $3t + 6g < n - k$ to let Σ have 3-multiplicative reconstruction. Thus, we only need

$$3t + k + 6\sqrt{q}^j < (q - \sqrt{q}) \cdot \sqrt{q}^{j-1}$$

for some $k = \epsilon n$ (ϵ is a constant), i.e.

$$t < \left(\frac{1}{3} - \frac{3}{2(\sqrt{q}-1)}\right) \cdot (1-\epsilon)n.$$

Take $t = n/4, q = 2^{10}, \epsilon = 1/10$, we get an $(n, n/4, n/10, 10)$-LSSS Σ.

Theorem 5. ([CC06]). *There exists an $(n, n/4, k, 10)$-LSSS Σ over $\mathbb{F}_2$ with $k = n/10 = \Theta(n)$ that has 3-multiplicative reconstruction.*

F.3 Instantiation of the Protocol

F.3.1 Functionalities for Preprocessing and Input.

In this section, we present the preprocessing functionality $\mathcal{F}_{\mathsf{prep}}$ and the input functionality $\mathcal{F}_{\mathsf{input}}$.

The preprocessing functionality $\mathcal{F}_{\mathsf{prep}}$ prepares the random sharings required for the executions of $\Pi_{\mathsf{Transpose}}$ together with the mask sharings of the $\Sigma^{(2)}$-sharings that need to be sent to clients for reconstructions.

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

On receiving (prep, C) from all the honest parties:

1. The trusted party generates an arithmetic circuit C' with the following properties:
   - For all input x, $C(x) = C'(x)$.
   - C' consists of an input layer, an output layer, D intermediate layers of addition/multiplication gates, where D is the depth of C. For each input wire of the circuit and output wire of an addition/multiplication gate, there is a fan-out gate taking this wire as input and copying it the number of times this wire will be used in future layers so that *every output wire of the input layer and the intermediate layers* is only used once as input wire in the later layers.
   - In the input layer and the output layer, the number of input wires attached to each client and the number of output wires attached to each client are multiples of k. In each intermediate layer, the number of addition gates and the number of multiplication gates are multiples of k^2. The number of input wires of the output layer is a multiple of k^2. The number of output wires of the fan-out gates in the input layer and each intermediate layer is also a multiple of k^2.
   - In each layer, k gates of the same type are grouped together (for input/output layers, k input/output wires attached to the same client are grouped together). Each group of output wires from the input layer and each intermediate layer serves as the input wires to a group of fan-out gates in this layer. The number of output wires of each group of fan-out gates is a multiple of k.
   - Circuit size: $|C'| = O(|C| + Dk^2 + mk)$, where m is the number of clients that provide inputs and D is the depth of C.

   Then the trusted party sends C' to all the parties.
2. For each honest client C_i and each batch of k output wires attached to client C_i in circuit C', the trusted party receives a set of corrupted servers' shares of a $\Sigma^{(2)}$-sharing from Sim.
3. For each honest client C_i and each batch of k output wires attached to client C_i in circuit C', the trusted party generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}]^{(2)}$ associated with this batch of output wires based on the corrupted servers' shares and distributes it to all the servers. Then the trusted party sends the secrets of these sharings to C_i.
4. For each corrupted client C_i and each batch of k output wires attached to client C_i in circuit C', the trusted party receives the secret $\boldsymbol{r}$ and a set of corrupted servers' shares of a $\Sigma^{(2)}$-sharing from Sim.
5. For each corrupted client C_i and each batch of k output wires attached to client C_i in circuit C', the trusted party generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}]^{(2)}$ associated with this batch of output wires based on the corrupted servers' shares and the secret $\boldsymbol{r}$. Then the trusted party distributes it to all the servers. Then the trusted party sends the secrets of these sharings to C_i.
6. Let W be the number of wires in C', do the following $10W/k^2$ times:

(a) For each honest server S_j, the trusted party receives a set of corrupted servers' shares of a $\Sigma^{(2)}$-sharing $[\boldsymbol{r}_j]^{(2)}$ from Sim.

(b) For each honest server S_j, the trusted party generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}_j]^{(2)}$ based on the corrupted servers' shares and distributes it to all the servers. Then the trusted party sends the secrets of this sharing to S_j.

(c) For each corrupted server S_j, the trusted party receives the secret $\boldsymbol{r}_j$ and a set of corrupted servers' shares of a $\Sigma^{(2)}$-sharing $[\boldsymbol{r}_j]^{(2)}$ from Sim.

(d) For each corrupted server S_j, the trusted party generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}_j]^{(2)}$ based on the corrupted servers' shares and the secret $\boldsymbol{r}_j$. Then the trusted party distributes it to all the servers. Then the trusted party sends the secrets of this sharing to S_j.

(e) The trusted party receives a set of corrupted servers' shares of $n\ell$ $\Sigma^{(2)}$-sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ from Sim.

(f) The trusted party generates $n\ell$ random $\Sigma^{(2)}$-sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ based on the corrupted servers' shares and distributes it to all the servers.

If abort is received from Sim, the trusted party sends abort to all the parties and aborts the functionality.

Figure 10: The preprocessing functionality.

The input functionality $\mathcal{F}_{\mathsf{input}}$ helps the clients to generate random Σ-sharings for their input.

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

On receiving $(\mathsf{input}, C', x_i)$ from each client C_i:

1. For each batch of k input wires attached to client C_i in circuit C', the trusted parties receive a set of corrupted servers' shares of a Σ-sharing from Sim.
2. For each batch of k input wires attached to client C_i in circuit C' with input values $s_1, \ldots, s_k \in \mathbb{F}_2$ (obtained from x_i), the trusted party randomly generates $[\boldsymbol{s}]$ based on corrupted servers' shares and the secret, and then the trusted party distributes $[\boldsymbol{s}]$ to all the servers, where $\boldsymbol{s} = (s_1, \ldots, s_k)$.

If abort is received from Sim, the trusted party sends abort to all the parties and aborts the functionality.

Figure 11: The input functionality.

We refer the readers to Section H for the instantiations of these two functionalities.

F.3.2 Subprotocols.

Now, we present the remaining subprotocols that are used in our construction of Π_0. Recall that the subprotocols $\Pi_{\mathsf{Transpose}}$ and Π_{Multi} have been presented in Section 6.

Sharing Transformations. A sharing transformation transforms a Σ-sharing $[\boldsymbol{x}]$ to a Σ-sharing of a linear map L of $\boldsymbol{s}$, i.e. $[L(\boldsymbol{x})]$. We also do k sharing transformations together with respect to k linear maps $L_1, \ldots, L_k$. This can be done with $O(n^2)$-bit communication by the following protocol Π_{Tran}.

Protocol Π_{Tran}

Input: The servers input public linear maps $L_1, \ldots, L_k : \mathbb{F}^k \to \mathbb{F}^k$ and their shares of Σ-sharings $[\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_k]$,

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

3. Let $\mathcal{M}$ be a $k \times k$ matrix defined by

$$\mathcal{M} = \begin{pmatrix} L_1(\boldsymbol{x}_1) \\ L_2(\boldsymbol{x}_2) \\ \vdots \\ L_k(\boldsymbol{x}_k) \end{pmatrix}.$$

Since $L_1, \ldots, L_k$ are linear, there exist constant vectors $\boldsymbol{c}_1^{(j)}, \ldots, \boldsymbol{c}_k^{(j)} \in \mathbb{F}^k$ such that

$$\mathcal{M}_{[:,j]} = \sum_{i=1}^{k} \boldsymbol{c}_i^{(j)} * \boldsymbol{x}_i^*$$

for $j = 1, \ldots, k$. The servers agree on public Σ-sharings $\{[\boldsymbol{c}_i^{(j)}]\}_{1 \le i,j \le k}$. Then the servers locally compute

$$[\mathcal{M}_{[:,j]}]^{(2)} = \sum_{i=1}^{k} [\boldsymbol{c}_i^{(j)}] \otimes [\boldsymbol{x}_i^*]$$

for $j = 1, \ldots, k$.

4. The servers run $\Pi_{\mathsf{Transpose}}$ with input sharings $[\mathcal{M}_{[:,1]}]^{(2)}, \ldots, [\mathcal{M}_{[:,k]}]^{(2)}$ and get output sharings $[L_1(\boldsymbol{x}_1)], \ldots, [L_k(\boldsymbol{x}_k)]$.

Figure 12: Protocol for sharing transformations.

Secret Collections from Different Positions. Secret collection is used to generate a Σ-sharing whose secret is collected from different Σ-sharings. With the help of the sharing transformation technique, we only need to handle a simple condition of secret collection where the secret is collected from k different positions of k sharings. If the input sharings are $[\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_k]$, we will generate Σ-sharing whose secret consists of the first entry of $\boldsymbol{x}_1$, the second entry of $\boldsymbol{x}_2$, ..., and the k-th entry of $\boldsymbol{x}_k$. Similar to multiplication gates and sharing transformations, we do k such collections together. This can be done with $O(n^2)$-bit communication by the following protocol Π_{Collect}.

Protocol Π_{Collect}

Input: The servers input their shares of $[\boldsymbol{x}_{1,i}], \ldots, [\boldsymbol{x}_{k,i}]$ for $i = 1, \ldots, k$ where each $\boldsymbol{x}_{j,i} = (x_{j,i}^{(1)}, \ldots, x_{j,i}^{(k)})$. Let $\boldsymbol{y}_i = (x_{1,i}^{(1)}, \ldots, x_{k,i}^{(k)})$ for each $i = 1, \ldots, k$.

1. For $i = 1, \ldots, k$, the servers locally compute

$$[\boldsymbol{y}_i]^{(2)} = [(x_{1,i}^{(1)}, \ldots, x_{k,i}^{(k)})]^{(2)} = \sum_{j=1}^{k} [\boldsymbol{e}_j] \otimes [\boldsymbol{x}_{j,i}],$$

where each $[\boldsymbol{e}_j]$ is a public Σ-sharing of a unit vector with the j-th bit equal to 1.

2. The servers run $\Pi_{\mathsf{Transpose}}$ with input sharings $[\boldsymbol{y}_1]^{(2)}, \ldots, [\boldsymbol{y}_k]^{(2)}$ and get output sharings $[\boldsymbol{y}_1^*], \ldots, [\boldsymbol{y}_k^*]$.
3. The servers locally compute $[\boldsymbol{y}_j^*]^{(2)} = [\mathbf{1}] \otimes [\boldsymbol{y}_j^*]$ for each $j = 1, \ldots, k$.
4. The servers run $\Pi_{\mathsf{Transpose}}$ with input sharings $[\boldsymbol{y}_1^*]^{(2)}, \ldots, [\boldsymbol{y}_k^*]^{(2)}$ and get output sharings $[\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_k]$.

Figure 13: Protocol to do secret collections from different positions.

Circuit Preprocessing. Then, we provide the circuit preprocessing protocol Π_{Circprep} as follows.

Protocol Π_{Circprep}

1. Recall that in C', gates that have the same type in each layer are divided into groups of size k. During the later computation, a batch of k wire values is stored in a single Σ-sharing. All parties determine how the wire values should be packed as follows:
   - For the input layer, for each group of k input gates belonging to the same client, the values of the output wires will be stored in a single Σ-sharing.
   - For each group of fan-out gates in the input layer, suppose it takes the wires $(w_1, \ldots, w_k)$ and vector $(n_1, \ldots, n_k)$ as input. Recall that in C', $(w_1, \ldots, w_k)$ are the output wires of a group of gates, n_i is the number of times we need to make copies of w_i, and $n_1 + \cdots + n_k$ is a multiple of k. All parties determine how the output wires should be packed:
     (a) Each party initiates an empty list $\mathcal{L}$. From $i = 1$ to k, each party inserts n_i times of w_i into L.
     (b) Let $h = (n_1 + \cdots + n_k)/k$. From $i = 1$ to h, the i-th output Σ-sharing will contain the values of wires $\mathcal{L}[(i-1) \cdot k + 1], \ldots, \mathcal{L}[i \cdot k]$.
   - For all computation layers, for each group of k multiplication gates or addition gates:
     * The values of the first input wires of these gates will be stored in a single Σ-sharing.
     * The values of the second input wires of these gates will be stored in a single Σ-sharing.
     * The values of the output wires of these gates will be stored in a single Σ-sharing.
   - For each group of fan-out gates in the intermediate layers, the wire values are packed in the same way as those for each fan-out gate in the input layer.
   - For the output layer, for each group of k output gates belonging to the same client, the values of the input wires will be stored in a single Σ-sharing.
2. Let N denote the number of output sharings of the input layer and all intermediate layers. Then the number of input sharings of the output layer and all intermediate layers is also N. The output sharings are labeled by $1, \ldots, N$, and the input sharings are also labeled by $1, \ldots, N$. The servers follow [GPS21] to attach a permutation p_i on the i-th output sharings.

Figure 14: Protocol for the circuit preprocessing.

F.3.3 Main Protocol.

Then, we provide our main protocol Π_0 as follows.

Protocol Π_0

1. Let C be the circuit to compute. The clients and servers sends (prep, C) to $\mathcal{F}_{\mathsf{prep}}$ and the receives:
   - A circuit C'.
   - A random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}]^{(2)}$ associated with each batch of output gates in C', and the receiver of the output of each group of gates get the secrets of this sharing.
   - $10W/k^2$ groups of:
     * Random $\Sigma^{(2)}$-sharings $[\boldsymbol{r}_1]^{(2)}, \ldots, [\boldsymbol{r}_n]^{(2)}$, where each server S_j holds $\boldsymbol{r}_j$.
     * Random $\Sigma^{(2)}$-sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$.

     Each group of sharings is associated with an execution of $\Pi_{\mathsf{Transpose}}$, where $\Pi_{\mathsf{Transpose}}$ will be executed for no more than $10W/k^2$ times in Π_0.
2. Each client C_i sends $(\mathsf{input}, C', x_i)$ to $\mathcal{F}_{\mathsf{input}}$, where x_i is the input of C_i.
3. The parties run the evaluation phase as follows.

Evaluation Phase

(a) The servers agree on a public Σ-sharing $[\mathbf{1}]$ where $\mathbf{1} = (1, 1, \ldots, 1)$. Then each Σ-sharing $[\boldsymbol{s}]$ can be locally converted to a $\Sigma^{(2)}$-sharing $[\boldsymbol{s}]^{(2)} = [\mathbf{1}] \otimes [\boldsymbol{s}]$. For $\boldsymbol{e}_1 = (1, 0, 0, \ldots, 0)$, $\boldsymbol{e}_2 = (0, 1, 0, \ldots, 0)$, $\ldots$, $\boldsymbol{e}_k = (0, 0, \ldots, 0, 1)$, all the servers agree on public Σ-sharings $[\boldsymbol{e}_1], \ldots, [\boldsymbol{e}_k]$.

(b) The clients and servers run Π_{Circprep}.

(c) All parties evaluate the circuit layer by layer as follows:

i. **Handling Fan-out Gates:** For each output sharings $[\boldsymbol{x}]$ of a group of gates (or input wires) in the previous layer, let n_i denote the number of times that the i-th secret of $\boldsymbol{x}$ is used in later layers. Then, the fan-out gates for $\boldsymbol{x}$ computes $(n_1 + \cdots + n_k)/k$ linear transformations on $\boldsymbol{x}$. For each k group of k linear transformations $L_1, \ldots, L_k$ on $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$, the servers can run Π_{Tran} to compute the output $[L_1(\boldsymbol{x}_1)], \ldots, [L_k(\boldsymbol{x}_k)]$ (as we discussed in Section 6). Note that the number of output wires of all the fan-out gates in each layer is a multiple of k^2, which matches that each Π_{Tran} can compute sharings on k^2 output wires of fan-out gates.

ii. **Permuting the Secrets:** For each k output sharings $[\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_k]$ of the previous layer, let $p_1, \ldots, p_k$ denote the permutations (expressed as linear maps from $\mathbb{F}_2^k$ to $\mathbb{F}_2^k$) associated with them. All the servers run Π_{Tran} with input $[\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_k]$ and $p_1, \ldots, p_k$. Then the secrets of the input sharings in the next layer come from different positions in the output sharings of previous layers.

iii. **Collecting Secrets from Previous Layers:** For each k input sharings $[\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_k]$, let $[\boldsymbol{x}_j^{(i)}]$ denote the output sharing from previous layers whose i-th secret is the i-th secret of $q_j(\boldsymbol{x}_j)$ where q_j is a permutation. Since each of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ comes from different positions in the output sharings of the previous layers, there exists output sharings $[\boldsymbol{x}_j^{(1)}], \ldots, [\boldsymbol{x}_j^{(k)}]$ from the output sharings of previous layers for each $j = 1, \ldots, k$. Then, all servers run Π_{Collect} with input sharings $[\boldsymbol{x}_j^{(1)}], \ldots, [\boldsymbol{x}_j^{(k)}]$ for $j = 1, \ldots, k$.

iv. **Permuting the Secrets:** For each k sharings $[q_1(\boldsymbol{x}_1)], \ldots, [q_k(\boldsymbol{x}_k)]$ collected by Π_{Collect}, all the servers run Π_{Tran} with input $[q_1(\boldsymbol{x}_1)], \ldots, [q_k(\boldsymbol{x}_k)]$ and $q_1^{-1}, \ldots, q_k^{-1}$. Then the servers obtain the input sharings $[\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_k]$ of this layer.

v. **Evaluating Multiplication Gates and Addition Gates:** For each k group of multiplication gates with input sharings $([\boldsymbol{x}_1], [\boldsymbol{y}_1]), \ldots, ([\boldsymbol{x}_k], [\boldsymbol{y}_k])$, all parties run Π_{Multi} with input sharings $[\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_k]$ and $[\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_k]$. For each group of addition gates with input sharings $[\boldsymbol{x}], [\boldsymbol{y}]$, all parties locally compute $[\boldsymbol{x} + \boldsymbol{y}] = [\boldsymbol{x}] + [\boldsymbol{y}]$. This step doesn't need to be done for the output layer.

(d) 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 15: The instantiation of protocol Π_0.

Theorem 6. *Protocol Π_0 satisfies the requirements listed in Section 4.1.*

We provide a proof of this theorem in Section G.

F.4 Cost Analysis for Π_0

We first analyze the communication cost of the evaluation phase of Π_0.

1. The step of running Π_{Circprep} only contains local computation and requires no communication.

2. While **Handling Fan-out Gates**, the communication cost comes from executions of Π_{Tran}. The sum of input wires of all the layers is bounded by $O(|C'|)$, so the number of sharing transformations we need to apply on the output sharings of all the layers is bounded by $O(|C'|/k) = O(|C'|/n)$, which requires $O(|C'|/n^2)$ executions of Π_{Tran}. Since the communication cost of Π_{Tran} is $O(n^2)$ bits, the communication cost of **Handling Fan-out Gates** is $O(|C'|)$ bits.

3. While **Permuting the Secrets** for the first time for each layer, the communication cost also comes from executions of Π_{Tran}. The sum of input wires of all the layers is bounded by $O(|C'|)$, so the number of permutations we need to apply on the output wires of all the layers is bounded by $O(|C'|/k) = O(|C'|/n)$, which requires $O(|C'|/n^2)$ executions of Π_{Tran}. Since the communication cost of Π_{Tran} is $O(n^2)$ bits, the communication cost of these executions of **Permuting the Secrets** is $O(|C'|)$ bits.

4. While **Collecting Secrets from Previous Layers**, the communication cost comes from executions of Π_{Collect}. The sum of input wires of all the layers is bounded by $O(|C'|)$, so we need to collect secrets from at most $O(|C'|)$ sharings, which requires $O(|C'|/n^2)$ executions of Π_{Collect}. Since the communication cost of Π_{Collect} is $O(n^2)$ bits, the communication cost of **Collecting Secrets from Previous Layers** is $O(|C'|)$ bits.

5. While **Permuting the Secrets** for the second time for each layer, the communication cost comes from executions of Π_{Tran}. The sum of input wires of all the layers is bounded by $O(|C'|)$, so the number of permutations we need to apply on the output wires of all the layers is bounded by $O(|C'|/k) = O(|C'|/n)$, which requires $O(|C'|/n^2)$ executions of Π_{Tran}. Since the communication cost of Π_{Tran} is $O(n^2)$ bits, the communication cost of these executions of **Permuting the Secrets** is also $O(|C'|)$ bits.

6. While **Evaluating Multiplication Gates and Addition Gates**, the communication cost comes from executions of Π_{Multi}. There are $O(|C'|/n)$ groups of multiplication gates to be computed, which requires $O(|C'|/n^2)$ executions of Π_{Multi}. Since the communication cost of Π_{Multi} is $O(n^2)$ bits, the communication cost of **Evaluating Multiplication Gates and Addition Gates** is $O(|C'|)$ bits.

7. While sending output sharings to the clients, the servers send a $\Sigma^{(2)}$-sharing of size $O(n)$ for each group of $O(n)$ output wires in C'. Thus, the communication cost is also bounded by $O(|C'|)$ bits.

To sum up, the total communication cost of the evaluation phase of Π_0 is $\mathsf{CC}_{\mathsf{eval}}^{\Pi_0} = O(|C'|) = O(|C| + Dn^2 + mn)$.

Note that the output size DS of $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the number G_A of AND gates in the servers' local circuit also affect the communication cost of Π_1', we also need to figure out how large DS and G_A are.

- **Data Size:** $\mathcal{F}_{\mathsf{prep}}$ outputs $O(|C'|/n)$ random $\Sigma^{(2)}$-sharings (of size $O(n)$). The total size of these sharings is $O(|C'|)$. $\mathcal{F}_{\mathsf{input}}$ outputs the input Σ-sharing (of size $O(n)$) for each batch of input wires ($O(|C'|/n)$ batches of input wires in toal). The total size of these sharings is $O(|C'|)$. Thus, the data size is $\mathsf{DS} = O(|C'|)$.

- **AND Gates:** During each server's local computation in the evaluation phase, only executions of Π_{Multi} contain non-linear operations. Since $k^2 = O(n^2)$ multiplication gates in C' are computed together in a single execution of Π_{Multi}, the servers need to run Π_{Multi} $O(|C'|/n^2)$ times. During each execution of Π_{Multi}, each party locally computes k tensor products on his shares of two Σ-sharings, which requires $k\ell^2 = O(n)$ computation of AND gates. For all the n servers, the number of AND gates computed during each execution of Π_{Multi} is $O(n^2)$. Thus, the total number of AND gates in the servers' local circuits is $G_A = O(|C'|)$.

Therefore, as analyzed in Section E, if Π_0 is instantiated by the protocol in Figure 15, the communication cost of Π_1' will be

$$\mathsf{CC}^{\Pi_1'} = \mathsf{CC}_{\mathsf{prep}} + \mathsf{CC}_{\mathsf{input}} + O(|C'|\kappa) = \mathsf{CC}_{\mathsf{prep}} + \mathsf{CC}_{\mathsf{input}} + O((|C| + Dn^2 + mn) \cdot \kappa).$$

G Proof of Theorem 6

Proof. It's easy to see that Π_0 meets the first three requirements listed in Section 4.1, so it remains to construct an ideal adversary Sim_0 that meets our requirement. We will show that the output in the ideal world is computationally indistinguishable from that in the real world using hybrid arguments. Our simulation is in the client-server model where the adversary corrupts any number of clients and exactly t servers.

We give the ideal adversary Sim_0 below.

Simulator Sim_0

1. Sim_0 emulates $\mathcal{F}_{\mathsf{prep}}$ to receive (prep, C) and the output shares and sharings to corrupted parties from $\mathcal{A}$. If abort is received from $\mathcal{A}$, Sim_0 aborts the protocol. After completing the simulation, Sim_0 outputs the adversary's view. Otherwise, Sim_0 faithfully emulates $\mathcal{F}_{\mathsf{prep}}$ to send C' and the output shares and sharings to the corrupted parties.
2. Sim_0 emulates $\mathcal{F}_{\mathsf{input}}$ to receive $(\mathsf{input}, C', x_i)$ for each corrupted client C_i and the output to corrupted parties from $\mathcal{A}$. If abort is received from $\mathcal{A}$, Sim_0 aborts the protocol. After completing the simulation, Sim_0 outputs the adversary's view. Otherwise, Sim_0 faithfully emulates $\mathcal{F}_{\mathsf{input}}$ to send C' and the output to the corrupted parties.
3. Sim_0 obtains the corrupted clients' input in the last step. Then, Sim_0 sends the corrupted clients' input to $\mathcal{F}$ and receives the corrupted clients' output.
4. Sim_0 simulates the evaluation phase of Π_0 as follows:

Evaluation Phase

(a) Sim_0 follows the protocol to get public sharings $[\mathbf{1}]$ and $[\boldsymbol{e}_1], \ldots, [\boldsymbol{e}_k]$ and run Π_{prepcirc}.

(b) Sim_0 simulates the evaluation process layer by layer as follows:

i. **Handling Fan-out Gates:** For each execution of Π_{Tran}:

A. Sim_0 follows the protocol to compute corrupted servers' shares of $[\boldsymbol{x}_j]^{(2)}$ for each $j = 1, \ldots, k$ using the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$.

B. For the first execution of $\Pi_{\mathsf{Transpose}}$:

1) For each $i = 1, \ldots, n$, Sim_0 follows the protocol to compute corrupted servers' shares of $[\boldsymbol{y}_i]^{(2)}$ (in $\Pi_{\mathsf{Transpose}}$) based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them.
2) Sim_0 randomly samples the whole sharing $[\boldsymbol{y}_i]^{(2)}$ based on the corrupted servers' shares.
3) For each corrupted server S_i, Sim_0 sends the honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ to S_i on behalf of the honest servers and reconstructs $\boldsymbol{y}_i$ for S_i.
4) Sim_0 follows the protocol to compute the corrupted servers' shares of $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$ based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and secrets of $\Sigma^{(2)}$-sharings reconstructed for them.

C. Sim_0 follows the protocol to compute corrupted servers' shares of $[\mathcal{M}_{[:,j]}]^{(2)}$ for $j = 1, \ldots, k$ based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them.

D. Sim_0 simulates the second execution of $\Pi_{\mathsf{Transpose}}$ as in Step B. to generate the $\Sigma^{(2)}$-sharing to be reconstructed and obtain the corrupted servers' shares of $([L_1(\boldsymbol{x}_1)], \ldots, [L_k(\boldsymbol{x}_k)])$.

ii. **Permuting the Secrets:** Sim_0 simulates each execution of Π_{Tran} in this step as in **Handling Fan-out Gates**.

iii. **Collecting Secrets from Previous Layers:** For each execution of Π_{Collect}:

A. Sim_0 follows the protocol to compute corrupted servers' shares of $[\boldsymbol{y}_i]^{(2)}$ for each $j = 1, \ldots, k$ based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them.

B. For the first execution of $\Pi_{\mathsf{Transpose}}$:

1) For each $i = 1, \ldots, n$, Sim_0 follows the protocol to compute corrupted servers' shares of $[\boldsymbol{y}_i]^{(2)}$ (in $\Pi_{\mathsf{Transpose}}$) based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them.
2) Sim_0 randomly samples the whole sharing $[\boldsymbol{y}_i]^{(2)}$ based on the corrupted servers' shares.
3) For each corrupted server S_i, Sim_0 Sim_0 sends the honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ (in $\Pi_{\mathsf{Transpose}}$) to S_i on behalf of the honest servers and reconstructs $\boldsymbol{y}_i$ for S_i.
4) Sim_0 follows the protocol to compute the corrupted servers' shares of $([\boldsymbol{y}_1^*], \ldots, [\boldsymbol{y}_k^*])$ (in Π_{Collect}) based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them.

C. Sim_0 follows the protocol to compute corrupted servers' shares of $[\boldsymbol{y}_j^*]^{(2)}$ for $j = 1, \ldots, k$ based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them.

D. Sim_0 simulates the second execution of $\Pi_{\mathsf{Transpose}}$ as in Step B. to generate the $\Sigma^{(2)}$-sharing to be reconstructed and obtain the corrupted servers' shares of $([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_k])$.

iv. **Permuting the Secrets:** Sim_0 simulates each execution of Π_{Tran} in this step as in **Handling Fan-out Gates**.

v. **Evaluating Multiplication Gates and Addition Gates:** For each group of addition gates, Sim_0 follows the protocol to compute the corrupted servers' shares of the Σ-sharings for output wires. For each execution of Π_{Multi}:

A. Sim_0 follows the protocol to compute corrupted servers' shares of $[\boldsymbol{z}_j]^{(2)}$ for each $j = 1, \ldots, k$ based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them.

B. For the first execution of $\Pi_{\mathsf{Transpose}}$:

1) For each $i = 1, \ldots, n$, Sim_0 follows the protocol to compute corrupted servers' shares of $[\boldsymbol{y}_i]^{(2)}$ (in $\Pi_{\mathsf{Transpose}}$) based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them.
2) Sim_0 randomly samples the whole sharing $[\boldsymbol{y}_i]^{(2)}$ based on the corrupted servers' shares.
3) For each corrupted server S_i, Sim_0 sends the honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ (in $\Pi_{\mathsf{Transpose}}$) to S_i on behalf of the honest servers and reconstructs $\boldsymbol{y}_i$ for S_i.
4) Sim_0 follows the protocol to compute the corrupted servers' shares of $([\boldsymbol{z}_1^*], \ldots, [\boldsymbol{z}_k^*])$ based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them.

C. Sim_0 follows the protocol to compute corrupted servers' shares of $[\boldsymbol{z}_j^*]^{(2)}$ for $j = 1, \ldots, k$ based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them.

D. Sim_0 simulates the second execution of $\Pi_{\mathsf{Transpose}}$ as in Step B. to generate the $\Sigma^{(2)}$-sharing to be reconstructed and obtain the corrupted servers' shares of $([\boldsymbol{z}_1], \ldots, [\boldsymbol{z}_k])$.

(c) After simulating the evaluation of all the layers of the circuit and the collection of the input sharings of the output gates, for each input sharing $[\boldsymbol{y}]$ for an output gate attached to each client C_i:

- If C_i is honest, Sim_0 follows the protocol to compute the corrupted servers' shares of $[\boldsymbol{y} + \boldsymbol{r}]^{(2)}$ based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them. Then, Sim_0 randomly samples the whole sharing $[\boldsymbol{y} + \boldsymbol{r}]^{(2)}$ based on the corrupted servers' shares.
- If C_i is corrupted, Sim_0 follows the protocol to compute the corrupted servers' shares of $[\boldsymbol{y}]$. Then Sim_0 randomly samples the whole sharing $[\boldsymbol{y}]$ based on the secret and corrupted servers' shares. Then Sim_0 follows the protocol to computes the whole sharing $[\boldsymbol{y} + \boldsymbol{r}]^{(2)}$ based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them.

5. Sim_0 outputs the view of the adversary as well as the honest servers' shares of all the $\Sigma^{(2)}$-sharings that are sent for reconstructions in the evaluation phase.

Figure 16: The simulator for Π_0.

We construct the following hybrids:

Hyb$_0$: In this hybrid, Sim_0 gets honest clients' inputs and runs the protocol honestly. This corresponds to the real-world scenario.

Hyb$_1$: In this hybrid, after emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$, Sim_0 regards that the corrupted parties output from the two functionalities are just the received outputs of them from $\mathcal{A}$. Then Sim_0 follows the protocol of the evaluation phase to run the corrupted parties. For each $\Sigma^{(2)}$-sharing whose receiver is a corrupted server S_i, Sim_0 reconstructs the secret for S_i. This doesn't affect the output distribution. Thus, **Hyb**$_1$ and **Hyb**$_0$ have the same output distribution.

Hyb$_2$: In this hybrid, while **Handling Fan-out Gates** during the evaluation phase, for the first execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Tran}, Sim_0 doesn't follow the protocol to compute honest servers'

shares of $[\boldsymbol{y}_i]^{(2)}$ for each honest server S_i. Instead, Sim_0 first computes corrupted servers' shares based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them and then randomly samples the whole sharing based on the corrupted servers' shares. Then Sim_0 computes the whole sharing $[\boldsymbol{r}_i]^{(2)}$ by $[\boldsymbol{y}_i]^{(2)} - [F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell})]^{(2)}$ for each honest server S_i. Similarly, Sim_0 doesn't follow the protocol to compute honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ for each corrupted server S_i. Instead, Sim_0 computes the secret of this sharing first and then randomly samples the whole sharing based on the corrupted servers' shares (still based on the received output from $\mathcal{A}$ while emulating $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and the secrets of $\Sigma^{(2)}$-sharings reconstructed for them) and the secret. Then Sim_0 computes the whole sharing $[\boldsymbol{r}_i]^{(2)}$ by $[\boldsymbol{y}_i]^{(2)} - [F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell})]^{(2)}$ for each corrupted server S_i.

Since for each honest server S_i, $[\boldsymbol{r}_i]^{(2)}$ is generated randomly by $\mathcal{F}_{\mathsf{prep}}$ based on the corrupted servers' shares received from $\mathcal{A}$, $[\boldsymbol{y}_i]^{(2)}$ is also completely random when corrupted servers' shares are fixed. Similarly, for each corrupted server S_i, $[\boldsymbol{r}_i]^{(2)}$ is generated randomly by $\mathcal{F}_{\mathsf{prep}}$ based on the corrupted servers' shares, $[\boldsymbol{y}_i]^{(2)}$ is also completely random when corrupted servers' shares and the secret are fixed. Therefore, we only change the order of generating the honest servers' shares of each pair of $[\boldsymbol{r}_i]^{(2)}$ and $[\boldsymbol{y}_i]^{(2)}$ without changing their distributions. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

Note that for each server S_i, the honest servers' shares of $[\boldsymbol{r}_i]^{(2)}$ in these executions of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids.

$\mathbf{Hyb}_3$: In this hybrid, while **Handling Fan-out Gates** during the evaluation phase, for the first execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Tran}, Sim_0 doesn't follow the protocol to compute each server S_i's shares of $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$ by $\boldsymbol{y}_i - \boldsymbol{r}_i$. Instead, Sim_0 computes them by $F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell})$. Since $[\boldsymbol{r}_i]^{(2)}$ is computed by $[\boldsymbol{y}_i]^{(2)} - [F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell})]^{(2)}$, $\boldsymbol{y}_i - \boldsymbol{r}_i = F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell})$ always holds. Therefore, we only change the way of generating each server S_i's shares of $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$ without changing their distributions. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same output distribution.

Note that for each honest server S_i, the secrets $\boldsymbol{r}_i$ in these executions of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids.

$\mathbf{Hyb}_4$: In this hybrid, while **Handling Fan-out Gates** during the evaluation phase, for the first execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Tran}, Sim_0 doesn't compute each server S_i's shares of $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$ by $F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell})$. Instead, Sim_0 randomly generates the value of $\boldsymbol{y}_i = F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell}) + \boldsymbol{r}_i$ for each corrupted server S_i and then use it to compute S_i's shares of $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$ by $\boldsymbol{y}_i - \boldsymbol{r}_i$. Then, Sim_0 samples the honest servers' shares of $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$ based on the secrets and the corrupted parties' shares. Note that for each honest server S_i, the honest servers' shares and the secrets of the random sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ associated with this execution of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids.

Since $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell}$ are all sampled randomly, the computation of $F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell})$ for each corrupted server S_i is just the generation process of $S-i$'s shares of some random Σ-sharings based on the secrets. Thus the corrupted servers' $F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell})$ are uniformly random, so $F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell}) + \boldsymbol{r}_i$ for each corrupted S_i is also uniformly random. Therefore, we only change the order of generating each corrupted server S_i's $F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell}) + \boldsymbol{r}_i$ and $F_i(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell})$ without changing their distributions. Besides, since $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{n\ell}$ are all sampled randomly, the honest servers' shares of $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$ are randomly sampled based on corrupted servers' shares and the secrets in both hybrids. Thus, $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ have the same output distribution.

$\mathbf{Hyb}_5$: In this hybrid, while **Handling Fan-out Gates** during the evaluation phase, for the second execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Tran}, Sim_0 doesn't follow the protocol to compute honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ for each server S_i. Instead, Sim_0 randomly samples the whole sharing based on the corrupted servers' shares. In addition, Sim_0 doesn't follow the protocol to compute the servers' shares of $([L_1(\boldsymbol{x}_1)], \ldots, [L_k(\boldsymbol{x}_k)])$. Instead, Sim_0 computes each corrupted server S_i's shares by $\boldsymbol{y}_i - \boldsymbol{r}_i$ and samples the honest servers' shares based on the secrets and the corrupted servers' shares. Note that the secrets and the honest servers' shares of $[\boldsymbol{r}_i]^{(2)}$ for each S_i and the random sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ associated with each execution of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids. For the same reason in $\mathbf{Hyb}_2$, $\mathbf{Hyb}_3$, and $\mathbf{Hyb}_4$, we conclude that $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same output distribution.

$\mathbf{Hyb}_6$: In this hybrid, while **Permuting the Secrets** for the first time during the evaluation phase for each layer, for the first execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Tran}, Sim_0 doesn't follow the protocol to compute honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ for each server S_i. Instead, Sim_0 randomly samples the whole sharing based on the corrupted servers' shares. In addition, Sim_0 doesn't follow the protocol to compute the servers' shares of $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$. Instead, Sim_0 computes each corrupted server S_i's shares by $\boldsymbol{y}_i - \boldsymbol{r}_i$ and randomly samples the honest servers' shares based on the secrets and the corrupted servers' shares. Note that the secrets and the honest servers' shares of $[\boldsymbol{r}_i]^{(2)}$ for each S_i and the random sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ associated with each execution of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids. For the same reason in $\mathbf{Hyb}_2$, $\mathbf{Hyb}_3$, and $\mathbf{Hyb}_4$, we conclude that $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ have the same output distribution.

$\mathbf{Hyb}_7$: In this hybrid, while **Permuting the Secrets** for the first time during the evaluation phase for each layer, for the second execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Tran}, Sim_0 doesn't follow the protocol to compute honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ for each server S_i. Instead, Sim_0 randomly samples the whole sharing based on the corrupted servers' shares. In addition, Sim_0 doesn't follow the protocol to compute servers' shares of $([L_1(\boldsymbol{x}_1)], \ldots, [L_k(\boldsymbol{x}_k)])$. Instead, Sim_0 computes each corrupted server S_i's shares by $\boldsymbol{y}_i - \boldsymbol{r}_i$ and randomly samples the honest servers' shares based on the secrets and the corrupted servers' shares. Note that the secrets and the honest servers' shares of $[\boldsymbol{r}_i]^{(2)}$ for each S_i and the random sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ associated with each execution of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids. For the same reason in $\mathbf{Hyb}_2$, $\mathbf{Hyb}_3$, and $\mathbf{Hyb}_4$, we conclude that $\mathbf{Hyb}_7$ and $\mathbf{Hyb}_6$ have the same output distribution.

Note that the honest parties' shares of $[\boldsymbol{s}]$ for each input Σ-sharing generated by an honest client are not used in the later simulation, Sim_0 doesn't generate them in future hybrids.

$\mathbf{Hyb}_8$: In this hybrid, while **Collecting Secrets from Previous Layers** during the evaluation phase, for the first execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Collect} while collecting secrets from previous layers in each layer, Sim_0 doesn't follow the protocol to compute honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ (in $\Pi_{\mathsf{Transpose}}$) for each server S_i. Instead, Sim_0 randomly samples the whole sharing based on the corrupted servers' shares. In addition, Sim_0 doesn't follow the protocol to compute honest servers' shares of $([\boldsymbol{y}_1^*], \ldots, [\boldsymbol{y}_k^*])$ (in Π_{Collect}). Instead, Sim_0 computes each corrupted server S_i's shares by $\boldsymbol{y}_i - \boldsymbol{r}_i$ and randomly samples the servers' shares based on the secrets and the corrupted servers' shares. Note that the secrets and the honest servers' shares of $[\boldsymbol{r}_i]^{(2)}$ for each S_i and the random sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ associated with each execution of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids. For the same reason in $\mathbf{Hyb}_2$, $\mathbf{Hyb}_3$, and $\mathbf{Hyb}_4$, we conclude that $\mathbf{Hyb}_8$ and $\mathbf{Hyb}_7$ have the same output distribution.

$\mathbf{Hyb}_9$: In this hybrid, while **Collecting Secrets from Previous Layers** during the evaluation phase, for the second execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Collect} while collecting secrets from previous layers in each layer, Sim_0 doesn't follow the protocol to compute honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ (in $\Pi_{\mathsf{Transpose}}$) for each server S_i. Instead, Sim_0 randomly samples the whole sharing based on the corrupted servers' shares. In addition, Sim_0 doesn't follow the protocol to compute the servers' shares of $([\boldsymbol{y}_1], \ldots, [\boldsymbol{y}_k])$ (in Π_{Collect}). Instead, Sim_0 computes each corrupted server S_i's shares by $\boldsymbol{y}_i - \boldsymbol{r}_i$ and randomly samples the honest servers' shares based on the secrets and the corrupted servers' shares. Note that the secrets and the honest servers' shares of $[\boldsymbol{r}_i]^{(2)}$ for each S_i and the random sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ associated with each execution of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids. For the same reason in $\mathbf{Hyb}_2$, $\mathbf{Hyb}_3$, and $\mathbf{Hyb}_4$, we conclude that $\mathbf{Hyb}_9$ and $\mathbf{Hyb}_8$ have the same output distribution.

Note that the honest servers' shares of the sharings $([\boldsymbol{y}_1^*], \ldots, [\boldsymbol{y}_k^*])$, $([\boldsymbol{y}_1^*]^{(2)}, \ldots, [\boldsymbol{y}_k^*]^{(2)})$, and $([\boldsymbol{y}_1]^{(2)}, \ldots, [\boldsymbol{y}_k]^{(2)})$ in each execution of Π_{Collect} while collecting secrets from previous layers are not used in the later simulation, Sim_0 doesn't generate them in future hybrids.

$\mathbf{Hyb}_{10}$: In this hybrid, while **Permuting the Secrets** for the second time during the evaluation phase for each layer, for the first execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Tran}, Sim_0 doesn't follow the protocol to compute honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ for each server S_i. Instead, Sim_0 randomly samples the whole sharing based on the corrupted servers' shares. In addition, Sim_0 doesn't follow the protocol to compute the servers' shares of $([\boldsymbol{x}_1^*], \ldots, [\boldsymbol{x}_k^*])$. Instead, Sim_0 computes each corrupted server S_i's shares by $\boldsymbol{y}_i - \boldsymbol{r}_i$ and randomly samples the honest servers' shares based on the secrets and the corrupted servers' shares. Note that

the secrets and the honest servers' shares of $[\boldsymbol{r}_i]^{(2)}$ for each S_i and the random sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ associated with each execution of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids. For the same reason in $\mathbf{Hyb}_2$, $\mathbf{Hyb}_3$, and $\mathbf{Hyb}_4$, we conclude that $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ have the same output distribution.

$\mathbf{Hyb}_{11}$: In this hybrid, while **Permuting the Secrets** for the second time during the evaluation phase for each layer, for the second execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Tran}, Sim_0 doesn't follow the protocol to compute honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ for each server S_i. Instead, Sim_0 randomly samples the whole sharing based on the corrupted servers' shares. In addition, Sim_0 doesn't follow the protocol to compute servers' shares of $([L_1(\boldsymbol{x}_1)], \ldots, [L_k(\boldsymbol{x}_k)])$. Instead, Sim_0 computes each corrupted server S_i's shares by $\boldsymbol{y}_i - \boldsymbol{r}_i$ and randomly samples the honest servers' shares based on the secrets and the corrupted servers' shares. Note that the secrets and the honest servers' shares of $[\boldsymbol{r}_i]^{(2)}$ for each S_i and the random sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ associated with each execution of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids. For the same reason in $\mathbf{Hyb}_2$, $\mathbf{Hyb}_3$, and $\mathbf{Hyb}_4$, we conclude that $\mathbf{Hyb}_{11}$ and $\mathbf{Hyb}_{10}$ have the same output distribution.

$\mathbf{Hyb}_{12}$: In this hybrid, while **Evaluating Multiplication Gates and Addition Gates** during the evaluation phase, for the first execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Multi} while evaluating the multiplication gates in each layer, Sim_0 doesn't follow the protocol to compute honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ (in Π_{Collect}) for each server S_i. Instead, Sim_0 randomly samples the whole sharing based on the corrupted servers' shares. In addition, Sim_0 doesn't follow the protocol to compute the servers' shares of $([\boldsymbol{z}_1^*], \ldots, [\boldsymbol{z}_k^*])$. Instead, each corrupted server S_j's shares by $\boldsymbol{y}_i - \boldsymbol{r}_i$ Note that the secrets and the honest servers' shares of $[\boldsymbol{r}_i]^{(2)}$ for each S_i and the random sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ associated with each execution of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids. For the same reason in $\mathbf{Hyb}_2$, $\mathbf{Hyb}_3$, and $\mathbf{Hyb}_4$, we conclude that $\mathbf{Hyb}_{12}$ and $\mathbf{Hyb}_{11}$ have the same output distribution.

$\mathbf{Hyb}_{13}$: In this hybrid, while **Evaluating Multiplication Gates and Addition Gates** during the evaluation phase, for the second execution of $\Pi_{\mathsf{Transpose}}$ in each execution of Π_{Multi} while evaluating the multiplication gates in each layer, Sim_0 doesn't follow the protocol to compute honest servers' shares of $[\boldsymbol{y}_i]^{(2)}$ (in $\Pi_{\mathsf{Transpose}}$) for each server S_i. Instead, Sim_0 randomly samples the whole sharing based on the corrupted servers' shares. In addition, Sim_0 doesn't follow the protocol to compute the servers' shares of $([\boldsymbol{z}_1], \ldots, [\boldsymbol{z}_k])$. Instead, each corrupted server S_j's shares by $\boldsymbol{y}_i - \boldsymbol{r}_i$ Note that the secrets and the honest servers' shares of $[\boldsymbol{r}_i]^{(2)}$ for each S_i and the random sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$ associated with each execution of $\Pi_{\mathsf{Transpose}}$ are not used in the later simulation, Sim_0 doesn't generate them in future hybrids. For the same reason in $\mathbf{Hyb}_2$, $\mathbf{Hyb}_3$, and $\mathbf{Hyb}_4$, we conclude that $\mathbf{Hyb}_{13}$ and $\mathbf{Hyb}_{12}$ have the same output distribution.

Note that the honest servers' shares of the sharings $([\boldsymbol{z}_1^*], \ldots, [\boldsymbol{z}_k^*])$, $([\boldsymbol{z}_1^*]^{(2)}, \ldots, [\boldsymbol{z}_k^*]^{(2)})$, and $([\boldsymbol{z}_1]^{(2)}, \ldots, [\boldsymbol{z}_k]^{(2)})$ in each execution of Π_{Multi} while evaluating the multiplication gates are not used in the later simulation, Sim_0 doesn't generate them in future hybrids.

$\mathbf{Hyb}_{14}$: In this hybrid, during the evaluation phase, for each k input wires of output gates attached to an honest client C_i, Sim_0 doesn't follow the protocol to compute the honest servers' shares of $[\boldsymbol{y} + \boldsymbol{r}]^{(2)}$. Instead, Sim_0 samples the whole sharing $[\boldsymbol{y} + \boldsymbol{r}]^{(2)}$ based on corrupted servers' shares. Then Sim_0 computes the sharing $[\boldsymbol{r}]^{(2)}$ by $[\boldsymbol{y} + \boldsymbol{r}]^{(2)} - [\boldsymbol{1}] \otimes [\boldsymbol{y}]$. Since $[\boldsymbol{r}]^{(2)}$ is generated randomly by $\mathcal{F}_{\mathsf{prep}}$ based on the corrupted servers' shares, $[\boldsymbol{y} + \boldsymbol{r}]^{(2)}$ is also completely random when corrupted servers' shares are fixed. Therefore, we only change the order of generating the honest servers' shares of each pair of $[\boldsymbol{r}]^{(2)}$ and $[\boldsymbol{y} + \boldsymbol{r}]^{(2)}$ without changing their distributions. Thus, $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{13}$ have the same output distribution.

$\mathbf{Hyb}_{15}$: In this hybrid, during the evaluation phase, for each k input wires of output gates attached to a corrupted client C_i, Sim_0 doesn't follow the protocol to compute the honest servers' shares of $[\boldsymbol{y} + \boldsymbol{r}]^{(2)}$. Instead, Sim_0 samples a random $\Sigma^{(2)}$ sharing as $[\boldsymbol{y} + \boldsymbol{r}]^{(2)}$ based on corrupted parties' shares and the secrets. Then, Sim_0 computes the honest servers' shares of $[\boldsymbol{r}]^{(2)}$ by $[\boldsymbol{y} + \boldsymbol{r}]^{(2)} - [\boldsymbol{1}] \otimes [\boldsymbol{y}]$. Since $[\boldsymbol{r}]^{(2)}$ is generated randomly by $\mathcal{F}_{\mathsf{prep}}$ based on the corrupted servers' shares and the secret, $[\boldsymbol{y} + \boldsymbol{r}]^{(2)}$ is also completely random when corrupted servers' shares are fixed. Therefore, we only change the order of generating the honest servers' shares of each pair of $[\boldsymbol{r}]^{(2)}$ and $[\boldsymbol{y} + \boldsymbol{r}]^{(2)}$ without changing their distributions. In addition, honest clients don't compute their output by themselves. Instead, honest clients get their output from $\mathcal{F}$. Since the computation process of $\mathcal{F}$ of $\boldsymbol{y}$ of each client (either honest or corrupted) is the computation of circuit C,

which has the same output as C', and the computation process of $[\boldsymbol{y}]$ by the servers is just the computation of C' in an additively-shared form. Thus, we only change the way to generate $\boldsymbol{y}$ without changing its distribution. Thus, $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$ have the same output distribution.

$\mathbf{Hyb}_{16}$: In this hybrid, during the evaluation phase, Sim_0 doesn't generate the honest servers' shares of the Σ-sharings for intermediate wire values. Note that these shares are not used in the simulation, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{16}$ and $\mathbf{Hyb}_{15}$ have the same output distribution.

Note that $\mathbf{Hyb}_{16}$ is the ideal-world scenario, Π_0 satisfies the requirements listed in Section 4.1.

□

H Realizing the Functionalities

H.1 Realizing $\mathcal{F}_{\mathsf{prep}}$

Circuit Transformation. The transformation from C to C' can be done locally. We sketch the transformation process (slightly modified from [GPS21]) below:

1. We insert a virtual client C_0, who provides constant default inputs and collects the wires whose value will not be output to any clients. Each input wire value of C_0 is used once in the circuit.

2. For each intermediate layer, we insert at most k^2-1 multiplication gates and k^2-1 addition gates to make the numbers of multiplication gates and addition gates multiples of k^2. Each gate being added takes 2 inputs from C_0 and outputs to C_0 as well. The inputs are set to be 0. This step increases the circuit size by $O(Dk^2)$.

3. For each intermediate layer, we divide the multiplication gates and addition gates into groups of k respectively. For the output wires of each group of gates, suppose that the k output wires are used $n_1, \ldots, n_k$ times in later layers. We increase $n'_k \leq k-1$ output wires to C_0 that take the wire value of the k-th output wire of the group of gates such that $n_1 + \cdots + n_k$ is a multiple of k. The fan-out gates for them will copy the k output wires $n_1, \ldots, n_k$ times respectively. After increasing these output wires, suppose that all the output wires of the multiplication gates and addition gates in this layer are used α times, then we further increase $\alpha' \leq k^2-1$ output wires to C_0 that take the value of the last output wire of the gates in this layer such that $\alpha+\alpha'$ is a multiple of k^2. This step increases the circuit size by $O(|C| + Dk^2)$.

4. For each client C_i ($i \in \{1, \ldots, m\}$), we insert at most $k-1$ input wires attached to him with input wire value 0 such that the total number of input wires attached to C_i is a multiple of k. We also insert the same number of output wires attached to C_0 that collect these input wires. Similarly, we insert at most $k-1$ output wires attached to him such that the total number of output wires attached to C_i is a multiple of k. We also insert the same number of input wires with input wire value 0 attached to C_0 that directly output to C_i. This step increases the circuit size by $O(mk)$.

5. For the input layer, we divide the input wires attached to each client C_i ($i \in \{1, \ldots, m\}$) into groups of k. For each group of input wires, we follow step 3 to make the total number that these k wires are used in later layers a multiple of k. This step increases the circuit size by $O(|C| + mk)$.

6. Finally, we insert input wires attached to C_0 to make the total number that input wires of all clients are used in later layers a multiple of k^2. Note that after this step, the number of output wires in the input layer and each intermediate layer is a multiple of k^2, and the number of input wires in each intermediate layer is also a multiple of k^2. This implies that the number of input wires in the output layer is a multiple of k^2 as well. Since each client C_i ($i \in \{1, \ldots, m\}$) takes output of size a multiple of k, this implies that the output size of C_0 is also a multiple of k. This step increases the circuit size by $O(k^2)$.

Following the above process, we obtain a circuit C' as we desire. We state the theorem below.

Theorem 7. ([GPS21], modified). *Given an arithmetic circuit C with input coming from m clients, there exists an efficient algorithm that takes C as input and outputs an arithmetic circuit C' with the following properties:*

- *For all input x, $C(x) = C'(x)$.*
- *C' consists of an input layer, an output layer, D intermediate layers of addition/multiplication gates, where D is the depth of C. For each input wire of the circuit and output wire of an addition/multiplication gate, there is a fan-out gate taking this wire as input and copying it the number of times this wire will be used in future layers so that* every output wire of the input layer and the intermediate layers *is only used once as input wire in the later layers.*
- *In the input layer and the output layer, the number of input wires attached to each client and the number of output wires attached to each client are multiples of k. In each intermediate layer, the number of addition gates and the number of multiplication gates are multiples of k^2. The number of input wires of the output layer is a multiple of k^2. The number of output wires of the fan-out gates in the input layer and each intermediate layer is also a multiple of k^2.*
- *In each layer, k gates of the same type are grouped together (for input/output layers, k input/output wires attached to the same client are grouped together). Each group of output wires from the input layer and each intermediate layer serves as the input wires to a group of fan-out gates in this layer. The number of output wires of each group of fan-out gates is a multiple of k.*
- *Circuit size: $|C'| = O(|C| + Dk^2 + mk)$, where m is the number of clients that provide inputs and D is the depth of C.*

Preparing Random Sharings. To prepare random $\Sigma^{(2)}$-sharings, 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 $t = n/4$, $\mathcal{N}$ be the matrix

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

   where $1, b_1, \ldots, b_{n-1}$ are n different elements in $\mathbb{F}_{2^a}$. The servers locally compute

$$\begin{pmatrix} [\boldsymbol{r}_1]^{(2)}_a \\ [\boldsymbol{r}_2]^{(2)}_a \\ \vdots \\ [\boldsymbol{r}_{n-t}]^{(2)}_a \end{pmatrix} = \mathcal{N} \cdot \begin{pmatrix} [\boldsymbol{s}_1]^{(2)}_a \\ [\boldsymbol{s}_2]^{(2)}_a \\ \vdots \\ [\boldsymbol{s}_n]^{(2)}_a \end{pmatrix}.$$

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

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

Preparing Zero Sharings. To mask the sharings generated by corrupted servers, we also need to prepare some random $\Sigma^{(2)}$-sharings whose secrets are all-zero vectors. This process is similar to generating random sharings except that we need an extra check to verify that their secrets are all zero.

Protocol Π_{Zero}

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

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

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

Verification. After receiving the shares for the sharings, the servers need to do a verification on them to ensure that all these sharings are valid Σ and $\Sigma^{(2)}$-sharings. We do the verification in the standard functionality $\mathcal{F}_{\mathsf{Coin}}$-hybrid model to verify a random linear combination of them.

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

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

Figure 19: Functionality for generating a common coin.

The functionality $\mathcal{F}_{\mathsf{Coin}}$ can be instantiated by letting each party P_j share a degree-t Shamir sharing over $\mathbb{F}_{2^\kappa}$ of a randomly chosen secret to all the parties and let each party reconstruct the secret of the sum of them. Finally, the parties locally check whether the received sharing is valid. Thus, $\mathcal{F}_{\mathsf{Coin}}$ can be realized in 2 rounds with communication of $O(n^2\kappa)$ bits.

Then we give the description of Π_{ver} in Figure 20.

Protocol Π_{ver}

For Σ-sharings $[\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_{k_1}]$, $\Sigma^{(2)}$-sharings (excluding the sharings prepared by Π_{Zero}) $[\boldsymbol{x}'_1]^{(2)}, \ldots, [\boldsymbol{x}'_{k_2}]^{(2)}$, and $\Sigma^{(2)}$-sharings $[\boldsymbol{o}_1]^{(2)}, \ldots, [\boldsymbol{o}_{k_3}]^{(2)}$ prepared by Π_{Zero} that need to be verified, the parties embed each Σ-sharings into a $\Sigma_{\times\kappa}$-sharing and embed each $\Sigma^{(2)}$-sharings into a $\Sigma^{(2)}_{\times\kappa}$-sharing. Then the parties get $[\boldsymbol{x}_1]_\kappa, \ldots, [\boldsymbol{x}_{k_1}]_\kappa$, $[\boldsymbol{x}'_1]^{(2)}_\kappa, \ldots, [\boldsymbol{x}'_{k_2}]^{(2)}_\kappa$, and $[\boldsymbol{o}_1]^{(2)}_\kappa, \ldots, [\boldsymbol{o}_{k_3}]^{(2)}_\kappa$:

1. Each server S_i generates a random $\Sigma_{\times\kappa}$-sharing $[\boldsymbol{r}^{(i)}]_\kappa$, a random $\Sigma^{(2)}_{\times\kappa}$-sharing $[\boldsymbol{r}^{(i)'}]^{(2)}_\kappa$, and a random $\Sigma^{(2)}_{\times\kappa}$-sharing $[\boldsymbol{o}^{(i)}]^{(2)}_\kappa$ with an all-zero secret. Then S_i distributes them to all the servers.
2. The servers invoke $\mathcal{F}_{\mathsf{Coin}}$ to get $s \in \mathbb{F}_{2^\kappa}$. If abort is received, abort the protocol. Then the servers expand s to a vector $(s_1, \ldots, s_{k_1}, s'_1, \ldots, s'_{k_2}, s^{(1)}, \ldots, s^{(k_3)}) \in \mathbb{F}_{2^\kappa}^{k_1+k_2+k_3}$ via a public pseudorandom generator.
3. All the servers locally computes $[\tau]_\kappa = \sum_{j=1}^{k_1} s_j \cdot [\boldsymbol{x}_j]_\kappa + \sum_{i=1}^{n} [\boldsymbol{r}^{(i)}]_\kappa$, $[\tau']^{(2)}_\kappa = \sum_{j=1}^{k_2} s'_j \cdot [\boldsymbol{x}'_j]^{(2)}_\kappa + \sum_{i=1}^{n} [\boldsymbol{r}^{(i)'}]^{(2)}_\kappa$, and $[\tau_0]^{(2)}_\kappa = \sum_{j=1}^{k_3} s^{(j)} \cdot [\boldsymbol{o}_j]_\kappa + \sum_{i=1}^{n} [\boldsymbol{o}^{(i)}]^{(2)}_\kappa$.
4. Each server sends his shares of $[\tau]_\kappa, [\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$ to all the servers.
5. Each server checks whether $[\tau]_\kappa$ is a valid $\Sigma_{\times\kappa}$-sharing, whether $[\tau']^{(2)}_\kappa$ is a valid $\Sigma^{(2)}_{\times\kappa}$-sharing, and whether $[\tau_0]^{(2)}_\kappa$ is a valid $\Sigma^{(2)}_{\times\kappa}$-sharing with an all-zero secret. If not, abort the protocol.

Figure 20: Verification of the sharings.

Summary. We provide the preprocessing protocol in Figure 21.

Protocol Π_{prep}

Input: A public circuit C.

1. All servers transform C to C'.
2. Let W be the number of wires in C', the servers run $\Pi_{\mathsf{RandShare}}$ to generate $10Wn\ell/k^2$ random $\Sigma^{(2)}$-sharings.
3. The servers run Π_{Zero} to prepare $10Wn/k^2 + W_O/k$ random $\Sigma^{(2)}$-sharings with all-zero secrets, where W_O is the number of output wires in C'.
4. For each batch of k output wires attached to client C_i in circuit C', C_i generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}']^{(2)}$ and distributes it to all the servers. Then the servers add a random $\Sigma^{(2)}$-sharing with an all-zero secret prepared by Π_{Zero} to this sharing and set the result to be $[\boldsymbol{r}]^{(2)}$.
5. The servers do the following $10W/k^2$ 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. Then the servers add a random $\Sigma^{(2)}$-sharings with an all-zero secret prepared by Π_{Zero} to this sharing and set the result to be $[\boldsymbol{r}_j]^{(2)}$.
   (b) The servers group each $n\ell$ random $\Sigma^{(2)}$-sharings obtained from $\Pi_{\mathsf{RandShare}}$ together as $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$.
6. The servers run Π_{ver} to verify all the $\Sigma^{(2)}$-sharings (including those from Π_{Zero}) generated in the preprocessing protocol, where each $\Sigma^{(2)}_{\times a}$-sharing is regarded as a $\Sigma^{(2)}$-sharings.

Figure 21: Realizing $\mathcal{F}_{\mathsf{prep}}$.

Theorem 8. *Let Σ be an (n, t, k, ℓ)-LSSS over $\mathbb{F}_2$ with 3-multiplicative reconstruction. Protocol Π_{prep} securely realizes $\mathcal{F}_{\mathsf{prep}}$ in the $\mathcal{F}_{\mathsf{Coin}}$-hybrid model against a fully malicious adversary that corrupts any number of clients and at most t servers.*

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

Before we give the construction of the ideal adversary, we recall that from Lemma 1, the honest servers' shares of any $\Sigma, \Sigma^{(2)}$-sharing (or a $\Sigma^{(2)}_{\times a}$-sharing which could be regarded as a group of a $\Sigma^{(2)}$-sharings) can uniquely determine the secret of the sharing.

We give the ideal adversary Sim below.

Simulator Sim

1. Sim sets $\mathsf{Check} = 0$.
2. Sim follows the protocol to transform C to C'.
3. For each execution of $\Pi_{\mathsf{RandShare}}$:
   (a) For each honest server S_i, Sim randomly generates corrupted servers' shares of $[\boldsymbol{s}_i]_a^{(2)}$ (using the first algorithm in Remark 1, same below) and sends them to the corrupted servers on behalf of S_i.
   (b) For each corrupted server S_i, Sim receives the honest servers' shares of the $\Sigma_{\times a}^{(2)}$-sharing $[\boldsymbol{s}_i]_a^{(2)}$. If the honest servers' shares are not from any valid $\Sigma_{\times a}^{(2)}$-sharing, Sim sets $\mathsf{Check} = 1$. Otherwise, Sim reconstructs the secret of this sharing and sets this sharing to be $[\boldsymbol{s}_i]_{\times a}^{(2)}$. From Lemma 1, the secret $\boldsymbol{r}$ unique.
   (c) Sim follows the protocol to compute corrupted servers' shares of $[\boldsymbol{r}_i^{(j)}]^{(2)}$ for $i = 1, \ldots, n-t, j = 1, \ldots, a$.
4. For each execution of Π_{Zero}:
   (a) For each honest server S_i, Sim randomly generates corrupted servers' shares of $[\boldsymbol{s}_i]_a^{(2)}$ and sends them to the corrupted servers on behalf of S_i.
   (b) For each corrupted server S_i, Sim receives the honest servers' shares of the $\Sigma_{\times a}^{(2)}$-sharing $[\boldsymbol{s}_i]_a^{(2)}$. If the honest servers' shares are not from any valid $\Sigma_{\times a}^{(2)}$-sharing, Sim sets $\mathsf{Check} = 1$. Otherwise, Sim reconstructs the secret of this sharing and sets this sharing to be $[\boldsymbol{s}_i]_{\times a}^{(2)}$. If the secret is not an all-zero vector, Sim sets $\mathsf{Check} = 1$.
   (c) Sim follows the protocol to compute corrupted servers' shares of $[\boldsymbol{o}_i^{(j)}]^{(2)}$ for $i = 1, \ldots, n-t, j = 1, \ldots, a$.
5. For each batch of k output wires attached to an honest client C_i in circuit C', Sim randomly generates corrupted servers' shares of the $\Sigma^{(2)}$-sharing $[\boldsymbol{r}']^{(2)}$ attached to this batch of wires and sends them to the corrupted servers on behalf of S_i. Then Sim follows the protocol to compute corrupted servers' shares of $[\boldsymbol{r}]^{(2)}$.
6. For each batch of k output wires attached to a corrupted client C_i in circuit C', Sim receives the honest servers' shares of the $\Sigma^{(2)}$-sharing $[\boldsymbol{r}']^{(2)}$ attached to this batch of wires. If the honest servers' shares are not from any valid $\Sigma^{(2)}$-sharing, Sim sets $\mathsf{Check} = 1$. Otherwise, Sim reconstructs the secret $\boldsymbol{r}$ of this sharing and sets this sharing to be $[\boldsymbol{r}']^{(2)}$. Then, Sim follows the protocol to compute corrupted servers' shares of $[\boldsymbol{r}]^{(2)}$.
7. Sim do the following $10W/k^2$ times in parallel:
   (a) For each honest server S_j, Sim randomly generates corrupted servers' shares of $[\boldsymbol{r}'_j]^{(2)}$ and distributes it to the corrupted servers on behalf of S_j. Then Sim follows the protocol to compute corrupted servers' shares of $[\boldsymbol{r}'_j]^{(2)}$.
   (b) For each corrupted server S_j, Sim receives the honest servers' shares of the $\Sigma^{(2)}$-sharing $[\boldsymbol{r}'_j]^{(2)}$ attached to this batch of wires. If the honest servers' shares are not from any valid $\Sigma^{(2)}$-sharing, Sim sets $\mathsf{Check} = 1$. Otherwise, Sim reconstructs the secret $\boldsymbol{r}'_j$ of this sharing and sets this sharing to be $[\boldsymbol{r}'_j]^{(2)}$. Then, Sim follows the protocol to compute corrupted servers' shares of $[\boldsymbol{r}_j]^{(2)}$.
   (c) Sim follows the protocol group each $n\ell$ random $\Sigma^{(2)}$-sharings obtained from $\Pi_{\mathsf{RandShare}}$ together as $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{n\ell}]^{(2)}$.
8. Sim simulate the execution of Π_{ver} as follows:
   (a) For each honest server S_i, Sim randomly generates corrupted servers' shares of $[\boldsymbol{r}^{(i)'}]_\kappa^{(2)}, [\boldsymbol{o}^{(i)}]_\kappa^{(2)}$ and sends them to the corrupted servers on behalf of S_i.
   (b) For each corrupted server S_i, Sim receives the honest servers' shares of the $\Sigma_{\times\kappa}^{(2)}$-sharing $[\boldsymbol{r}^{(i)'}]_\kappa^{(2)}, [\boldsymbol{o}^{(i)}]_\kappa^{(2)}$. If honest servers' shares of the sharing $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)'}]_\kappa^{(2)}$ or $\sum_{i\in\mathcal{C}}[\boldsymbol{o}^{(i)}]_\kappa^{(2)}$ ($\mathcal{C}$ is the set of indices of corrupted servers) are not from any valid $\Sigma_{\times\kappa}^{(2)}$-sharing, Sim sets $\mathsf{Check} = 1$. Otherwise, Sim reconstructs the secrets of the sharings and sets the sharings to be $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)'}]_\kappa^{(2)}, \sum_{i\in\mathcal{C}}[\boldsymbol{o}^{(i)}]_\kappa^{(2)}$. If the secret of the second sharing $\sum_{i\in\mathcal{C}}[\boldsymbol{o}^{(i)}]_\kappa^{(2)}$ is not an all-zero vector, Sim sets $\mathsf{Check} = 1$.
   (c) Sim emulates $\mathcal{F}_{\mathsf{Coin}}$ to receive RandCoin from all the corrupted servers and follows the protocol to sample $s \in \mathbb{F}_{2^\kappa}$ randomly.

(d) 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 all the corrupted servers and aborts the protocol. Let the pseudorandom coefficients for the $\Sigma^{(2)}$-sharings expanded from s be $s'_1, \ldots, s'_{k_2} \in \mathbb{F}_{2^\kappa}$. Let the pseudorandom coefficients for the $\Sigma^{(2)}$-sharings with all-zero secrets expanded from s be $s^{(1)}, \ldots, s^{(k_3)} \in \mathbb{F}_{2^\kappa}$.

(e)
- If $\mathsf{Check} = 0$:
  i. Sim follows the protocol to compute the corrupted servers' shares of $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$.
  ii. Sim generates honest servers' shares of $[\tau']^{(2)}_\kappa$ randomly based on corrupted servers' shares. Then Sim generates honest servers' shares of $[\tau_0]^{(2)}_\kappa$ randomly based on corrupted servers' shares and the all-zero secret.
  iii. Sim sends each honest server's share of $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$ to all the corrupted servers on behalf of this honest server.
  iv. Sim receives corrupted servers' shares of $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$ on behalf of each honest server and follows the protocol to check whether $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$ is valid $\Sigma^{(2)}_{\times\kappa}$-sharings and whether the secret of $[\tau_0]^{(2)}_\kappa$ is an all-zero vector. If not, Sim sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest server. Sim outputs the adversary's view after completing the simulation.
- If $\mathsf{Check} = 1$:
  i. For each $\Sigma^{(2)}, \Sigma^{(2)}_{\times a}, \Sigma^{(2)}_{\times\kappa}$ generated by an honest party, Sim generates the honest servers' shares randomly based on the corrupted servers' shares (and the all-zero secrets for those sharings generated in Π_{Zero}) and follows the protocol to compute honest servers' shares of $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$.
  ii. Sim follows the protocol to send each honest server's share of $[\tau]^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$ to all the corrupted servers.
  iii. Sim receives corrupted servers' shares of $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$ on behalf of each honest server and honestly check whether $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$ are valid $\Sigma^{(2)}_{\times\kappa}$-sharings and whether the secret of $[\tau_0]^{(2)}_\kappa$ is an all-zero vector. If not, Sim sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest server. Sim outputs the adversary's view after completing the simulation.
  iv. If the protocol is not aborted, Sim aborts the simulation.

9. Sim sends the corrupted servers' shares of all the output sharings of $\mathcal{F}_{\mathsf{prep}}$ to $\mathcal{F}_{\mathsf{prep}}$. For each batch of k output wires attached to a corrupted client C_i in circuit C', Sim sends the secret $\boldsymbol{r}$ of the sharing $[\boldsymbol{r}]^{(2)}$ attached to this batch of wires to $\mathcal{F}_{\mathsf{prep}}$.
10. Sim outputs the adversary's view.

Figure 22: The simulator for Π_{prep}.

We construct the following hybrids:

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

$\mathbf{Hyb}_1$: In this hybrid, whenever Sim generates a $\Sigma^{(2)}, \Sigma^{(2)}_{\times a}, \Sigma^{(2)}_{\times\kappa}$-sharing on behalf of an honest party, he first generates the corrupted servers' shares randomly and then generates the honest servers' shares based on the corrupted servers' shares and the secret. Since Σ is an (n, t, k, ℓ)-LSSS that has 3-multiplicative reconstruction, each t shares of a $\Sigma^{(2)}, \Sigma^{(2)}_{\times a}$, or a $\Sigma^{(2)}_{\times\kappa}$-sharing are uniformly random, so we only change the order of generating the honest servers' and the corrupted servers' shares. Thus, $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ have the same output distribution.

$\mathbf{Hyb}_2$: In this hybrid, Sim additionally sets $\mathsf{Check} = 0$ at the beginning of the simulation. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

$\mathbf{Hyb}_3$: In this hybrid, during each execution of $\Pi_{\mathsf{RandShare}}$, for each corrupted server S_i, on receiving the honest servers' shares of each $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{s}_i]^{(2)}_a$, Sim additionally checks whether the honest servers' shares are from a valid $\Sigma^{(2)}_{\times a}$-sharing. If not, Sim sets $\mathsf{Check} = 1$. Otherwise, Sim reconstructs the secret of this sharing and sets this sharing to be $[\boldsymbol{s}_i]^{(2)}_a$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same output distribution.

$\mathbf{Hyb}_4$: In this hybrid, during each execution of Π_{Zero}, for each corrupted server S_i, on receiving the honest servers' shares of each $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{s}_i]^{(2)}_a$, Sim additionally checks whether the honest servers' shares

are from a valid $\Sigma^{(2)}_{\times a}$-sharing with an all-zero secret. If not, Sim sets $\mathsf{Check} = 1$. Otherwise, Sim sets this sharing to be $[\boldsymbol{s}_i]^{(2)}_a$. 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, for each batch of k output wires attached to a corrupted client C_i in circuit C', on receiving the honest servers' shares of the $\Sigma^{(2)}$-sharing $[\boldsymbol{r}']^{(2)}$ attached to this batch of wires, Sim additionally checks whether the honest servers' shares are from a valid $\Sigma^{(2)}$-sharing. If not, Sim sets $\mathsf{Check} = 1$. Otherwise, Sim reconstructs the secret of this sharing and sets this sharing to be $[\boldsymbol{r}']^{(2)}$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same output distribution.

$\mathbf{Hyb}_6$: In this hybrid, for each corrupted server S_j, on receiving the honest servers' shares of each $\Sigma^{(2)}$-sharing $[\boldsymbol{r}'_j]^{(2)}$, Sim additionally checks whether the honest servers' shares are from a valid $\Sigma^{(2)}$-sharing. If not, Sim sets $\mathsf{Check} = 1$. Otherwise, Sim reconstructs the secret of this sharing and sets this sharing to be $[\boldsymbol{r}'_j]^{(2)}$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ have the same output distribution.

$\mathbf{Hyb}_7$: In this hybrid, while doing verification, for each corrupted server S_i, on receiving the honest servers' shares of each $\Sigma^{(2)}_{\times\kappa}$-sharing $[\boldsymbol{r}^{(i)'}]^{(2)}_\kappa, [\boldsymbol{o}^{(i)'}]^{(2)}_\kappa$, Sim additionally checks whether the honest servers' shares of $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)'}]^{(2)}_\kappa$ are from a valid $\Sigma^{(2)}_{\times\kappa}$-sharing and whether the honest servers' shares of $\sum_{i\in\mathcal{C}}[\boldsymbol{o}^{(i)}]^{(2)}_\kappa$ are from a valid $\Sigma^{(2)}_{\times\kappa}$-sharing with an all-zero secret. If not, Sim sets $\mathsf{Check} = 1$. Otherwise, Sim reconstructs the secrets of these sharings and sets this sharing to be $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)'}]^{(2)}_\kappa, \sum_{i\in\mathcal{C}}[\boldsymbol{o}^{(i)}]^{(2)}_\kappa$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_7$ and $\mathbf{Hyb}_6$ have the same output distribution.

$\mathbf{Hyb}_8$: In this hybrid, while doing verification, if $\mathsf{Check} = 0$, Sim doesn't follow the protocol to compute each honest server's share of $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$. Instead, he randomly samples honest servers' shares of $[\tau']^{(2)}_\kappa$ based on the corrupted servers' shares and randomly samples honest servers' shares of $[\tau_0]^{(2)}_\kappa$ based on the corrupted servers' shares and the all-zero secret. Then, the honest servers' shares of $[\boldsymbol{r}^{(j)'}]^{(2)}_\kappa, [\boldsymbol{o}^{(j)}]^{(2)}_\kappa$ for an honest server S_j are sampled based on $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$, the coefficients $s'_1, \ldots, s'_{k_2}, s^{(1)}, \ldots, s^{(k_3)}$, and all the $\Sigma^{(2)}$-sharings generated in the preprocessing protocol except each $[\boldsymbol{r}^{(j)'}]^{(2)}_\kappa, [\boldsymbol{o}^{(j)}]^{(2)}_\kappa$. Since $\mathsf{Check} = 0$, we can regard that corrupted parties follow the protocol to distribute all the $\Sigma^{(2)}$-sharings. Then, the sharing $[\tau']^{(2)}_\kappa$ must also be a valid $\Sigma^{(2)}_{\times\kappa}$-sharing. Since the honest servers' shares of $[\boldsymbol{r}^{(j)'}]^{(2)}_\kappa, [\boldsymbol{o}^{(j)}]^{(2)}_\kappa$ are sampled randomly based on the corrupted servers' shares, the honest servers' shares of $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$ are also random in the case that the corrupted servers' shares are fixed. Thus, we only change the order of generating the honest servers' shares of $[\boldsymbol{r}^{(j)'}]^{(2)}_\kappa, [\boldsymbol{o}^{(j)}]^{(2)}_\kappa$ and $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$ without changing their distributions. Thus, $\mathbf{Hyb}_8$ and $\mathbf{Hyb}_7$ have the same output distribution.

$\mathbf{Hyb}_9$: In this hybrid, while doing verification, if $\mathsf{Check} = 1$, Sim delays the generation of the honest servers' shares of each $\Sigma^{(2)}$-sharings (including $\Sigma^{(2)}_{\times a}, \Sigma^{(2)}_{\times\kappa}$-sharings). Sim generates them while computing the honest servers' shares of $[\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$. Since these sharings are not used in the simulation before computing the honest servers' shares of $[\tau']^{(2)}_\kappa$, this doesn't change the output distribution. Thus, $\mathbf{Hyb}_9$ and $\mathbf{Hyb}_8$ have the same output distribution.

$\mathbf{Hyb}_{10}$: In this hybrid, while doing verification, after following the protocol to check the $\Sigma^{(2)}$-sharings, Sim aborts the simulation if $\mathsf{Check} = 1$. This only changes the output distribution if $\mathsf{Check} = 1$ but the verification passes.

Since $\mathsf{Check} = 1$, suppose that not all of the sharings used to compute $[\tau']^{(\kappa)}_\kappa$ are sent correctly, i.e. there exists a $\Sigma^{(2)}$-sharing to be verified that is not distributed correctly or $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)'}]^{(2)}_\kappa$ is not a valid $\Sigma^{(2)}_{\times\kappa}$-sharing. If all the $\Sigma^{(2)}$-sharings to be verified are valid, but $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)'}]^{(2)}_\kappa$ is not a valid $\Sigma^{(2)}_{\times\kappa}$-sharing, then $\sum_{j=1}^{k_2} s'_j \cdot [\boldsymbol{x}'_j]^{(2)}_\kappa$ must be a valid $\Sigma^{(2)}_{\times\kappa}$-sharing, so $[\tau']^{(2)}_\kappa = \sum_{j=1}^{k_2} s'_j \cdot [\boldsymbol{x}'_j]^{(2)}_\kappa + \sum_{i=1}^{n}[\boldsymbol{r}^{(i)'}]^{(2)}_\kappa$ must not be a valid $\Sigma^{(2)}_{\times\kappa}$-sharing. Then, the verification can't pass.

Now we consider the case that a $\Sigma^{(2)}$-sharing to be verified that is not distributed correctly. 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 $[\tau]_\kappa = \sum_{j=1}^{k_1} s_j \cdot [\boldsymbol{x}_j]_\kappa + \sum_{i=1}^{n}[\boldsymbol{r}^{(i)}]_\kappa$ 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 $2^{-\kappa}$, which is negligible. Thus, if there is a non-negligible probability that $[\tau]_\kappa$ is valid, then the truly random field elements and the pseudo-random values $s_1, \ldots, s_{k_1} \in \mathbb{F}_{2^\kappa}$ can be distinguished by computing $[\tau]_\kappa$ with a non-negligible probability, which contradicts the definition of a PRG, so the probability that $[\tau]_\kappa$ is valid is negligible.

Similarly, the probability that not all of the sharings used to compute $[\tau_0]_\kappa^{(2)}$ are sent correctly but the verification passes is also negligible. Therefore, the distribution only changes with a negligible probability. Thus, the distributions of $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ are computationally indistinguishable.

$\mathbf{Hyb}_{11}$: In this hybrid, Sim doesn't generate the honest servers' shares of those sharings generated by honest parties when $\mathsf{Check} = 0$, and honest parties don't compute their output by themselves. Instead, Sim sends the corrupted parties' output to $\mathcal{F}_{\mathsf{prep}}$, and honest parties directly get their output from $\mathcal{F}_{\mathsf{prep}}$. Since when $\mathsf{Check} = 0$, the honest servers' shares of those sharings generated by honest parties are not used in the simulation if the honest parties directly get their output from $\mathcal{F}_{\mathsf{prep}}$, we only need to argue that the outputs of honest parties obtained in the two hybrids are of the same distribution.

For the $\Sigma^{(2)}$-sharings associated with output wires and each group of $([\boldsymbol{r}_1], \ldots, [\boldsymbol{r}_n])$, the only difference on the computation process of the honest parties' output is that $([\boldsymbol{r}_1], \ldots, [\boldsymbol{r}_n])$ is computed by adding a random sharing with an all-zero secret on the sharings $([\boldsymbol{r}_1'], \ldots, [\boldsymbol{r}_n'])$ with the same secrets as $([\boldsymbol{r}_1], \ldots, [\boldsymbol{r}_n])$. Thus, we only need to verify that the random sharings prepared in $\Pi_{\mathsf{RandShare}}$ are of the same distribution in both hybrids and verify that the random sharings with all-zero secrets are indeed random with corrupted servers' shares fixed. In this way, we can generate the honest servers' shares of $([\boldsymbol{r}_1], \ldots, [\boldsymbol{r}_n])$ randomly first based on the secrets and the corrupted parties' shares and then use them to compute the honest servers' shares of the sharings with all-zero secrets, and this won't change the output distribution.

Since each group of $\Sigma^{(2)}$-sharings $([\boldsymbol{r}_i^{(1)}]^{(2)}, \ldots, [\boldsymbol{r}_i^{(a)}]^{(2)})$ can be regard as a $\Sigma_{\times a}^{(2)}$-sharing $[\boldsymbol{r}_i]_a^{(2)}$, we can regard that Sim samples the sharing of $[\boldsymbol{r}_i]_a^{(2)}$ based on corrupted servers' shares. Let $\mathcal{C}$ be the set of corrupted servers' indices and $\mathcal{H}$ be the set of honest servers' indices. Then let $\mathcal{N}_\mathcal{C}$ denote the $(n-t) \times t$ 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

$$\begin{pmatrix} [\boldsymbol{r}_1]_a^{(2)} \\ [\boldsymbol{r}_2]_a^{(2)} \\ \vdots \\ [\boldsymbol{r}_{n-t}]_a^{(2)} \end{pmatrix} = \mathcal{N} \cdot \begin{pmatrix} [\boldsymbol{s}_1]_a^{(2)} \\ [\boldsymbol{s}_2]_a^{(2)} \\ \vdots \\ [\boldsymbol{s}_n]_a^{(2)} \end{pmatrix} = \mathcal{N}_\mathcal{C} \cdot \left([\boldsymbol{s}_j]_a^{(2)}\right)_{j\in\mathcal{C}} + \mathcal{N}_\mathcal{H} \cdot \left([\boldsymbol{s}_j]_a^{(2)}\right)_{j\in\mathcal{H}}.$$

Recall that $\mathcal{N}$ is a Vandermonde matrix of size $(n-t) \times n$. Therefore $\mathcal{N}_\mathcal{H}$ is a Vandermonde matrix of size $(n-t) \times (n-t)$, which is invertible. Thus, there is a bijective map from $\left([\boldsymbol{s}_j]_a^{(2)}\right)_{j\in\mathcal{H}}$ and $\left([\boldsymbol{r}_i]_a^{(2)}\right)_{i=1}^{n-t}$. Recall that $\left([\boldsymbol{s}_j]_a^{(2)}\right)_{j\in\mathcal{H}}$ are sampled randomly based on corrupted servers' shares and the secrets, so $\left([\boldsymbol{r}_i]_a^{(2)}\right)_{i=1}^{n-t}$ is also completely random when corrupted servers' shares are fixed. Therefore, the output distributions of the random sharings are the same in both hybrids.

Similarly, the random sharings with all-zero secrets are also completely random when corrupted servers' shares and the secrets are fixed. Thus, $\mathbf{Hyb}_{11}$ and $\mathbf{Hyb}_{10}$ have the same output distribution.

Note that $\mathbf{Hyb}_{11}$ is the ideal-world scenario, Π_{prep} computes $\mathcal{F}_{\mathsf{prep}}$ with computational security.

□

Cost Analysis. We analyze the communication cost of Π_{prep} step by step as follows (where we regard each $\Sigma_{\times a}^{(2)}$-sharing as a $\Sigma^{(2)}$-sharings):

1. Step 1 only contains local computation and requires no communication.

2. In Step 2, the servers generate $10Wn\ell/k^2 = O(|C'|/n)$ random $\Sigma^{(2)}$-sharings, where the random $\Sigma^{(2)}$-sharings are prepared with amortized cost $O(n)$. Thus, the communication cost of this step is $O(|C'|)$.

3. In Step 3, the servers generate $10Wn/k^2 + W_O/k = O(|C'|/n)$ random $\Sigma^{(2)}$-sharings with all-zero secrets, where the random $\Sigma^{(2)}$-sharings are prepared with amortized cost $O(n)$. Thus, the communication cost of this step is $O(|C'|)$.

4. In Step 4, the clients need to send a Σ-sharing of size $O(n)$ for each batch of $k = O(n)$ input wires. The communication is linear to the number W_I of input wires of C', i.e. $O(|W_I|) < O(|C'|)$ bits.

5. In Step 5, the servers distributes $10Wn/k^2 = O(|C'|/n)$ $\Sigma^{(2)}$-sharings of size $O(n)$. Thus, the communication cost of this step is $O(|C'|)$.

6. In Step 6, the instantiation of $\mathcal{F}_{\mathsf{Coin}}$ requires communication of $O(n^2\kappa)$ bits. The servers distributes $2n$ $\Sigma^{(2)}_{\times\kappa}$-sharings of size $O(n\kappa)$ in this step. Thus, the communication cost of this step is $O(n^2\kappa)$.

As analyzed above, the total communication of Π_{prep} is $\mathsf{CC}_{\mathsf{prep}} = O(|C'|+n^2\kappa) = O(|C|+Dn^2+mn+n^2\kappa)$.

H.2 Realizing $\mathcal{F}_{\mathsf{input}}$

To realize $\mathcal{F}_{\mathsf{input}}$, we let each client share their input values via Σ-sharings. Then the servers jointly check that the Σ-sharings are all valid.

Protocol Π_{input}

1. For each batch of k input wires attached to client C_i in circuit C' with input values $s_1, \ldots, s_k \in \mathbb{F}_2$, C_i randomly generates $[\boldsymbol{s}]$ and distributes it to all the servers, where $\boldsymbol{s} = (s_1, \ldots, s_k)$.
2. The servers run Π_{ver} to verify all the input Σ-sharings generated by the clients.

Figure 23: Realizing $\mathcal{F}_{\mathsf{input}}$.

Theorem 9. *Let Σ be an (n, t, k, ℓ)-LSSS over $\mathbb{F}_2$ with 3-multiplicative reconstruction. Protocol Π_{input} securely realizes $\mathcal{F}_{\mathsf{input}}$ in the $\mathcal{F}_{\mathsf{Coin}}$-hybrid model against a fully malicious adversary that corrupts any number of clients and at most t servers.*

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

We give the ideal adversary Sim below.

Simulator Sim

1. Sim sets $\mathsf{Check} = 0$.
2. For each batch of k input wires attached to an honest client C_i in circuit C', Sim randomly generates corrupted servers' shares of the input sharing $[\boldsymbol{s}]$ for this batch of wires and sends them to the corrupted servers on behalf of S_i.
3. For each batch of k input wires attached to a corrupted client C_i in circuit C', Sim receives the honest servers' shares of the input sharing $[\boldsymbol{s}]$ for this batch of wires. If the honest servers' shares are not from any valid Σ-sharing, Sim sets $\mathsf{Check} = 1$. Otherwise, Sim reconstructs the secret of this sharing and sets this sharing to be $[\boldsymbol{s}]$.
4. Sim simulate the execution of Π_{ver} as follows:
   (a) For each honest server S_i, Sim randomly generates corrupted servers' shares of $[\boldsymbol{r}^{(i)}]_\kappa$ (using the first algorithm in Remark 1, same below) and sends them to the corrupted servers on behalf of S_i.
   (b) For each corrupted server S_i, Sim receives the honest servers' shares of the $\Sigma_{\times\kappa}$-sharing $[\boldsymbol{r}^{(i)}]_\kappa$. If

honest servers' shares of the sharing $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)}]_\kappa$ ($\mathcal{C}$ is the set of indices of corrupted servers) are not from any valid $\Sigma_{\times\kappa}$-sharing, Sim sets $\mathsf{Check}=1$. Otherwise, Sim reconstructs the secret of this sharing and sets this sharing to be $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)}]_\kappa$.

(c) Sim emulates $\mathcal{F}_{\mathsf{Coin}}$ to receive $\mathsf{RandCoin}$ from all the corrupted servers and follows the protocol to samples $s\in\mathbb{F}_{2^\kappa}$ randomly.

(d) 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 all the corrupted servers and aborts the protocol. Let the pseudorandom coefficients for the Σ-sharings expanded from s be $s_1,\ldots,s_{k_1}\in\mathbb{F}_{2^\kappa}$.

(e)
- If $\mathsf{Check}=0$:
  i. Sim follows the protocol to compute the corrupted servers' shares of $[\tau]_\kappa$.
  ii. Sim generates honest servers' shares of $[\tau]_\kappa$ randomly based on corrupted servers' shares.
  iii. Sim sends each honest server's share of $[\tau]_\kappa$ to all the corrupted servers on behalf of this honest server.
  iv. Sim receives corrupted servers' shares of $[\tau]_\kappa$ on behalf of each honest server and follows the protocol to check whether $[\tau]_\kappa$ is a valid $\Sigma_{\times\kappa}$-sharing. If not, Sim sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest server. Sim outputs the adversary's view after completing the simulation.
- If $\mathsf{Check}=1$:
  i. For each $\Sigma,\Sigma_{\times\kappa}$ generated by an honest party, Sim generates the honest servers' shares randomly based on the corrupted servers' shares and follows the protocol to compute honest servers' shares of $[\tau]_\kappa$.
  ii. Sim follows the protocol to send each honest server's share of $[\tau]_\kappa$ to all the corrupted servers.
  iii. Sim receives corrupted servers' shares of $[\tau]_\kappa$ on behalf of each honest server and honestly check whether $[\tau]_\kappa$ is a valid $\Sigma_{\times\kappa}$-sharing. If not, Sim sends abort to $\mathcal{F}_{\mathsf{prep}}$ and aborts the protocol on behalf of the honest server. Sim outputs the adversary's view after completing the simulation.
  iv. If the protocol is not aborted, Sim aborts the simulation.

5. Sim sends the corrupted servers' shares of all the output sharings of $\mathcal{F}_{\mathsf{input}}$ to $\mathcal{F}_{\mathsf{input}}$. For each batch of k input wires attached to a corrupted client C_i in circuit C', Sim sends $[\boldsymbol{s}]$ for this batch of wires to $\mathcal{F}_{\mathsf{input}}$.
6. Sim outputs the adversary's view.

Figure 24: The simulator for Π_{input}.

We construct the following hybrids:

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

$\mathbf{Hyb}_1$: In this hybrid, whenever Sim generates a $\Sigma,\Sigma_{\times\kappa}$-sharing on behalf of an honest party, he first generates the corrupted servers' shares randomly and then generates the honest servers' shares based on the corrupted servers' shares and the secret. Since Σ is an (n,t,k,ℓ)-LSSS that has 3-multiplicative reconstruction, each t shares of a Σ-sharing or a $\Sigma_{\times\kappa}$-sharing are uniformly random, so we only change the order of generating the honest servers' and the corrupted servers' shares. Thus, $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ have the same output distribution.

$\mathbf{Hyb}_2$: In this hybrid, Sim additionally sets $\mathsf{Check}=0$ at the beginning of the simulation. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

$\mathbf{Hyb}_3$: In this hybrid, for each batch of k input wires attached to a corrupted client C_i in circuit C', on receiving the honest servers' shares of the input sharing $[\boldsymbol{s}]$ for this batch of wires, Sim additionally checks whether the honest servers' shares are from a valid Σ-sharing. If not, Sim sets $\mathsf{Check}=1$. Otherwise, Sim reconstructs the secret of this sharing and sets this sharing to be $[\boldsymbol{s}]$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same output distribution.

$\mathbf{Hyb}_4$: In this hybrid, while doing verification, for each corrupted server S_i, on receiving the honest servers' shares of each $\Sigma_{\times\kappa}$-sharing $[\boldsymbol{r}^{(i)}]_\kappa$, Sim additionally checks whether the honest servers' shares of $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)}]_\kappa$ are from a valid $\Sigma_{\times\kappa}$-sharing. If not, Sim sets $\mathsf{Check}=1$. Otherwise, Sim reconstructs the secret of this sharing and sets this sharing to be $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)}]_\kappa$. 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, while doing verification, if $\mathsf{Check} = 0$, Sim doesn't follow the protocol to compute each honest server's share of $[\tau]_\kappa$. Instead, he randomly samples honest servers' shares of $[\tau]_\kappa$ based on the corrupted servers' shares. Then, the honest servers' shares of $[\boldsymbol{r}^{(j)}]_\kappa$ for an honest server S_j are sampled based on $[\tau]_\kappa$, the coefficients $s_1, \ldots, s_{k_1}$, and all the Σ-sharings generated in the input protocol except $[\boldsymbol{r}^{(j)}]_\kappa$. Since $\mathsf{Check} = 0$, we can regard that corrupted parties follow the protocol to distribute all the Σ-sharings. Then, the sharing $[\tau]_\kappa$ must also be a valid $\Sigma_{\times\kappa}$-sharing. Since the honest servers' shares of $[\boldsymbol{r}^{(j)}]_\kappa$ are sampled randomly based on the corrupted servers' shares, the honest servers' shares of $[\tau]_\kappa$ are also random in the case that the corrupted servers' shares are fixed. Thus, we only change the order of generating the honest servers' shares of $[\boldsymbol{r}^{(j)}]_\kappa$ and $[\tau]_\kappa$ without changing their distributions. Thus, $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same output distribution.

$\mathbf{Hyb}_6$: In this hybrid, while doing verification, if $\mathsf{Check} = 1$, Sim delays the generation of the honest servers' shares of each Σ-sharings (including $\Sigma_{\times a}, \Sigma_{\times\kappa}$-sharings) by an honest party. Sim generates them while computing the honest servers' shares of $[\tau]_\kappa$. Since these sharings are not used in the simulation before computing the honest servers' shares of $[\tau]_\kappa$, this doesn't change the output distribution. Thus, $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ have the same output distribution.

$\mathbf{Hyb}_7$: In this hybrid, while doing verification, after following the protocol to check the Σ-sharings, Sim aborts the simulation if $\mathsf{Check} = 1$. This only changes the output distribution if $\mathsf{Check} = 1$ but the verification passes.

Since $\mathsf{Check} = 1$, there must be a Σ-sharing to be verified that is not distributed correctly or $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)}]_\kappa$ is not a valid $\Sigma_{\times\kappa}$-sharing. If all the Σ-sharings to be verified are valid, but $\sum_{i\in\mathcal{C}}[\boldsymbol{r}^{(i)}]_\kappa$ is not a valid $\Sigma_{\times\kappa}$-sharing, then $\sum_{j=1}^{k_1} s_j \cdot [\boldsymbol{x}_j]_\kappa$ must be a valid $\Sigma_{\times\kappa}$-sharing, so $[\tau]_\kappa = \sum_{j=1}^{k_1} s_j \cdot [\boldsymbol{x}_j]_\kappa + \sum_{i=1}^{n}[\boldsymbol{r}^{(i)}]_\kappa$ must not be a valid $\Sigma_{\times\kappa}$-sharing. Then, the verification can't pass.

Now we consider the case that a Σ-sharing to be verified that is not distributed correctly. 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 $[\tau]_\kappa = \sum_{j=1}^{k_1} s_j \cdot [\boldsymbol{x}_j]_\kappa + \sum_{i=1}^{n}[\boldsymbol{r}^{(i)}]_\kappa$ 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 $2^{-\kappa}$, which is negligible. Thus, if there is a non-negligible probability that $[\tau]_\kappa$ is valid, then the truly random field elements and the pseudo-random values $s_1, \ldots, s_{k_1} \in \mathbb{F}_{2^\kappa}$ can be distinguished by computing $[\tau]_\kappa$ with a non-negligible probability, which contradicts the definition of a PRG, so the probability that $[\tau]_\kappa$ is valid is negligible.

Therefore, the distribution only changes with a negligible probability. Thus, the distributions of $\mathbf{Hyb}_7$ and $\mathbf{Hyb}_6$ are computationally indistinguishable.

$\mathbf{Hyb}_8$: In this hybrid, Sim doesn't generate the honest servers' shares of those sharings generated by honest parties when $\mathsf{Check} = 0$, and honest parties don't compute their output by themselves. Instead, Sim sends the corrupted parties' output to $\mathcal{F}_{\mathsf{input}}$, and honest parties directly get their output from $\mathcal{F}_{\mathsf{input}}$. Since when $\mathsf{Check} = 0$, the honest servers' shares of those sharings generated by honest parties are not used in the simulation if the honest parties directly get their output from $\mathcal{F}_{\mathsf{input}}$, we only need to argue that the outputs of honest parties obtained in the two hybrids are of the same distribution. Since the computation process of the honest parties' output is the same in both hybrids, the outputs of honest parties obtained in the two hybrids are of the same distribution. Thus, $\mathbf{Hyb}_8$ and $\mathbf{Hyb}_7$ have the same output distribution.

Note that $\mathbf{Hyb}_8$ is the ideal-world scenario, Π_{input} computes $\mathcal{F}_{\mathsf{input}}$ with computational security.

□

Cost Analysis. In Π_{input}, for each group of $k = O(n)$ input wires of C', a client needs to distribute a Σ-sharing of $O(n)$ bits. Besides, each server needs to distribute a $\Sigma_{\times\kappa}$-sharing of size $n\kappa$ for the verification. Thus, the communication cost is $\mathsf{CC}_{\mathsf{input}} = O(W_I + n^2\kappa)$, where w_I is number of input wires in C'.

I Analysis of Rounds for Π'_1

We analyze the number of rounds we need in Π'_1 as follows:

1. **Preprocessing.** All the sharings need to be generated in Π_{prep} can be sent in parallel in one round. The instantiation of $\mathcal{F}_{\mathsf{Coin}}$ requires 2 rounds, where the first round of sending random Shamir sharings can be performed in parallel with the first round of the protocol. Then, the verification of the sharings can be done in one round of sending shares of $[\tau]_\kappa, [\tau']^{(2)}_\kappa, [\tau_0]^{(2)}_\kappa$. Then, the preprocessing step requires 3 rounds.

2. **Input.** Similar as Π_{prep}, the protocol Π_{input}, the input step also requires 3 rounds. Note that the input step and the preprocessing step can be performed in parallel, this step does not require extra rounds of communication.

3. **Generating Output Labels.** The generation of output labels only requires one round of sharing the Σ and $\Sigma^{(3)}$-sharings, which can also be performed in parallel with the preprocessing. Thus, this step does not require extra rounds of communication.

4. **Garbling Local Circuits.** This step requires one extra round of sending garbled circuits to P_{king}.

5. **Encrypting Input Labels.** This step requires one round of sending ciphertexts of input labels to P_{king}, which can be sent in parallel with the garbled circuit. Thus, this step does not require extra rounds of communication.

6. **Sending Input Labels.** This step requires one round of sending input labels of output from $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ to P_{king}, which can also be sent in parallel with the garbled circuit. Thus, this step does not require extra rounds of communication.

7. **Evaluating the Circuit.** This step only contains local computation.

8. **Sending Outputs.** This step contains one extra round of sending outputs to the clients.

As analyzed above, protocol Π'_1 requires 5 total rounds of communication.

J Dishonest Majority Constant-Round MPC

In this section, we transform the MPC protocol Π'_1 (instantiated by Π_0) from our constant-round MPC compiler to support the dishonest majority setting, where up to $t_s = (1-\epsilon)n$ ($\epsilon > 0$ is a constant) of n servers may be corrupted.

J.1 Subprotocols

Let $N = \Theta(n + \kappa)$ be the number of virtual servers, Π_0 be the non-constant round MPC that is secure against $N/4$ corruptions of N (virtual) servers constructed in Section F. Let Σ be the $(N, N/4, k, \ell)$-LSSS over $\mathbb{F}_2$ with 3-multiplicative reconstruction used in Π_0. We will construct a protocol Π_2 that runs among m clients and n servers with any number of corrupted clients and at most $(1-\epsilon)n$ corrupted servers. At the beginning of Π_2, all parties will together generate a random coin and use it to select N random committees, each of size $c = \frac{\log 32}{\epsilon}$, to emulate N virtual servers. From the analysis in Section 7, with an overwhelming probability that $15N/16$ virtual servers contain an honest real-world server.

Let $V_1, \ldots, V_N$ be all the virtual servers. For each $i \in \{1, \ldots, N\}$, the c real-world servers who emulate V_i are denoted by $S_{i,1}, \ldots, S_{i,c}$. Let $\langle a \rangle$ be the additive sharing of a among c servers. Our construction Π_2 is under the assumption of random OTs with message length $\kappa - 1$, a random oracle (RO) $\mathcal{O}_1$ with output length κ, and another RO $\mathcal{O}_2$ with output length $c\ell\kappa$.

First, we provide the standard functionalities of a random OT and a commitment scheme.

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

The trusted party interacts with two parties P_1, P_2.

1. The trusted party randomly samples two messages $r_0, r_1 \in \{0,1\}^{\kappa-1}$ and a bit $b \in \{0,1\}$. If P_1 is corrupted, r_0, r_1 are chosen by P_1. If P_2 is corrupted, b, r_b are chosen by P_2.
2. The trusted party sends (r_0, r_1) to P_1 and sends (b, r_b) to P_2.

Figure 25: Functionality for random oblivious transfer.

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

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

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

Figure 26: Functionality for commitment [DKL+13].

In the random oracle model, the functionality $\mathcal{F}_{\mathsf{Commit}}$ can be instantiated by letting P_i send the output $\mathcal{O}(i\|x\|\tau_x\|r_x)$ of a random oracle $\mathcal{O}$ to all the parties as the commitment of x (where r_x is a random value chosen by P_i). The size of each commitment is $O(\kappa)$. Then all the parties check whether P_i distributes the same commitment to all parties by exchanging the commitment with each other, which requires communication of $O(n^2\kappa)$ bits. If $\mathcal{F}_{\mathsf{Commit}}$ is invoked many times in parallel, the check can be done together by putting all the commitments together as input to the random oracle and cross-checking the result. In this way, the amortized communication cost of realizing $\mathcal{F}_{\mathsf{Commit}}$ can be reduced to $O(n\kappa)$ if it is invoked by all the n parties in parallel. To open the value x to P_j, P_i only needs to send x, r_x to P_j.

With such a commitment scheme, the functionality $\mathcal{F}_{\mathsf{Coin}}$ (Figure 19) can be instantiated in the dishonest majority setting by letting each party first send the commitment of a randomly sampled field element in $\mathbb{F}_{2^\kappa}$ to all the parties and then open it to all the parties. Then the the output of $\mathcal{F}_{\mathsf{Coin}}$ can be computed by adding all these values together. Note that the check of whether each party distributes the same commitment to all parties can be done in parallel with the opening of the commitments. Thus, the number of rounds required for $\mathcal{F}_{\mathsf{Coin}}$ is 2, and the communication cost of each invocation of $\mathcal{F}_{\mathsf{Coin}}$ is $O(n^2\kappa)$ (for $\mathcal{F}_{\mathsf{Coin}}$ invoked by n parties over $\mathbb{F}_{2^\kappa}$).

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

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

The protocol runs between c parties $P_1, \ldots, P_c$. 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 27: Protocol for opening the secret of an additive sharing.

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

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

The protocol runs between c parties $P_1, \ldots, P_c$. 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 28: Protocol for multiplication.

J.2 Protocol Description

Now we are ready to introduce our construction of Π_2, which runs the Sharing Phase $\Pi_{2\text{-Share}}$, the Local Computation Phase $\Pi_{2\text{-Local}}$, the Garbling Phase $\Pi_{2\text{-Garble}}$, the Verification Phase $\Pi_{2\text{-Ver}}$, and the Evaluation Phase $\Pi_{2\text{-Eval}}$ in order.

Sharing Phase. In the sharing phase, the parties distribute all the sharings that need to be generated in Steps 1-3 in Π_1' to the virtual servers. Besides, the servers invoke $\mathcal{F}_{\mathsf{ROT}}$ to get the additional preprocessing data for the local computation of virtual servers. By sampling the local randomness of each server in garbling the virtual servers' local circuits, the local computation of each virtual server is deterministic on all the data prepared in the sharing phase. At the end of the sharing phase, the servers commit their inputs and local randomness for the local computation of each virtual server.

Protocol $\Pi_{2\text{-Share}}$

Sharing Phase

Determining the Virtual Servers: All the parties call $\mathcal{F}_{\mathsf{Coin}}$ and get a random coin r in $\mathbb{F}_{2^\kappa}$. Based on the random coin r, the parties run a public PRG to agree on random c-server sets $\{S_{j,1}, \ldots, S_{j,c}\}$ for $j = 1, \ldots, N$, and the servers $S_{j,1}, \ldots, S_{j,c}$ will emulate a virtual server V_j in Π_0.

Let the local circuit of each virtual server V_j in Π_0 be $\mathtt{Circ}^{V_j}$. The parties do the following:

Emulating $\mathcal{F}_{\mathsf{prep}}$:

1. **Transforming the Circuit.** All the parties locally transform the circuit C to C' as in Π_0 (followed by Theorem 7 in Section H.1).
2. **Preparing Masks for Output Sharings.** For each batch of k output wires attached to client C_i in circuit C', C_i generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{r}']^{(2)}$ and distributes them to the virtual servers, i.e. additively distributes each share $\boldsymbol{r}^{V_j}$ of virtual server V_j to $S_{j,1}, \ldots, S_{j,c}$.
3. **Preparing Random Sharings.** Let $a = \lfloor \log n \rfloor + 1$. Each server S samples $10WN\ell/(ank^2\epsilon)$ random $\Sigma_{\times a}^{(2)}$-sharings (of the form $[\boldsymbol{s}]_a^{(2)}$) and distributes them to the virtual servers.
4. **Preparing Zero Sharings.** Let $a = \lfloor \log n \rfloor + 1$. Each server S samples $10WN/(ank^2\epsilon) + W_O/(ank\epsilon)$ random $\Sigma_{\times a}^{(2)}$-sharings (of the form $[\boldsymbol{s}]_a^{(2)}$ where $\boldsymbol{s}$ is an all-zero vector) with all-zero secrets and distributes them to the virtual servers.
5. **Preparing Masks for Transpose Protocols.** For each virtual server V_i and $j = 1, \ldots, c$, each server $S_{i,j}$ samples $10W/k^2$ random $\Sigma^{(2)}$-sharings (of the form $[\boldsymbol{r}_{i,j}]^{(2)}$) and distributes them to the virtual servers.
6. **Preprocessing for the Verification of Sharings.** Let $\kappa' = N + \kappa$. Each real-word server S_i generates a random $\Sigma_{\times\kappa'}$-sharing $[\boldsymbol{r}^{(i)}]_{\kappa'}$, a random $\Sigma_{\times\kappa'}^{(2)}$-sharing $[\boldsymbol{r}^{(i)'}]_{\kappa'}^{(2)}$, and a random $\Sigma_{\times\kappa'}^{(2)}$-sharing $[\boldsymbol{o}^{(i)}]_{\kappa'}^{(2)}$ with an all-zero secret. Then S_i distributes them to the virtual servers.

Emulating $\mathcal{F}_{\mathsf{input}}$: For each batch of k input wires attached to client C_i in circuit C' with input values $s_1, \ldots, s_k \in \mathbb{F}_2$, C_i randomly generates $[\boldsymbol{s}]$, where $\boldsymbol{s} = (s_1, \ldots, s_k)$. Then C_i distributes the sharing to the virtual servers.

Preparing for the Garbling of Local Circuits:

1. **Calling $\mathcal{F}_{\mathsf{ROT}}$.** For each virtual server V_j, each pair of servers $(S_{j,\alpha}, S_{j,\beta})$, $S_{j,\beta}$ call $\mathcal{F}_{\mathsf{ROT}}$ $(4c+1)\cdot G_A^{V_j} + 2c\ell^3\mathsf{rec}$ times, where $G_A^{V_j}$ is the number of AND gates in $\mathtt{Circ}^{V_j}$ (recall that rec is the number of reconstructions of $\Sigma^{(2)}$-sharings in Π_0). Each time $S_{j,\alpha}$ receives $r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)}$ and $S_{j,\beta}$ receives $b^{(\alpha,\beta)}, r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)}$.
2. **Preparing for the Output Labels.** For $i = 1, \ldots, \mathsf{rec}$, if the receiver R_i of the i-th reconstruction of Π_0 is a server, let the corresponding virtual server be V_j. For $\alpha = 1, \ldots, \kappa$ and $\beta = 1, \ldots, c$, each $S_{j,\beta}$ generates a random $\Sigma^{(3)}$-sharing as $[\boldsymbol{r}_{\mathbf{0},\beta}^{(\alpha)}]^{(3)}$ and a random Σ-sharing as $[\boldsymbol{r}_{\mathbf{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0},\beta}^{(\alpha)}]$. Then $S_{j,\beta}$ distributes each pair of $[\boldsymbol{r}_{\mathbf{0},\beta}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\mathbf{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0},\beta}^{(\alpha)}]$ to the virtual servers. If R_i is a client C_j, he plays as c servers of a virtual server to generate and distribute the sharings.
3. **Generating Local Randomness.** For each virtual server V_j with local circuit $\mathtt{Circ}^{V_j}$, the servers $S_{j,1}, \ldots, S_{j,c}$ do the following:
   (a) Each server $S_{j,i}$ samples a random $(\kappa-1)$-bit string as $\Delta^{S_{j,i}}$.
   (b) For each wire w that is not an output wire of an XOR gate or an output gate, each server $S_{j,i}$ samples a random bit $\lambda_w^{S_{j,i}}$ as his share of $\langle\lambda_w\rangle$ and a random $(\kappa-1)$-bit string as $k_{w,0}^{S_{j,i}}$.

Committing Local Inputs: For each $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharing generated by a party P in the sharing phase, for each share s for a virtual server V_j, the servers $S_{j,1}, \ldots, S_{j,c}$ holds $\langle s\rangle$. Let the set of all these shares, all the output from $\mathcal{F}_{\mathsf{ROT}}$, and all the local randomness generated in the last step of a server $S_{j,i}$ be $\mathsf{IS}^{S_{j,i}}$. $S_{j,i}$ sends $(\mathsf{commit}, S_{j,i}, \mathsf{IS}^{S_{j,i}}, \tau_{\mathsf{IS}^{S_{j,i}}})$ to $\mathcal{F}_{\mathsf{Commit}}$. For the remaining steps of the protocol, $\mathsf{IS}^{S_{j,i}}$ is regarded as $S_{j,i}$'s committed input to the local computation of V_j.

Figure 29: The sharing phase of the dishonest majority protocol Π_2.

Local Computation Phase. In the local computation phase, the real-world servers emulate the virtual servers to do the local linear computation to generate preprocessing data for the garbling (i.e. computing the output of $\mathcal{F}_{\mathsf{prep}}$ in Π_0). Concretely, for the computation of $a \oplus b$ of a virtual server V_j, the servers $S_{j,1}, \ldots, S_{j,c}$ computes $\langle a\oplus b\rangle = \langle a\rangle \oplus \langle b\rangle$. The local computation phase only contains local computation of real-world servers.

Protocol $\Pi_{2\text{-Local}}$

Local Computation Phase

For each virtual server V_j with local circuit $\mathtt{Circ}^{V_j}$ in Π_0, the servers $S_{j,1}, \ldots, S_{j,c}$ do the following:

1. **Computing Random Sharings.** We group the random sharings in Step 3 of the Sharing Phase (while emulating $\mathcal{F}_{\mathsf{prep}}$) into $10WN\ell/(ank^2\epsilon)$ groups, where the servers receive $\langle \boldsymbol{s}_i^{V_j}\rangle$ for $i = 1, \ldots, n$ in each group (each $\langle \boldsymbol{s}_i^{V_j}\rangle$ is generated by server S_i). Then:
   (a) Let $\mathcal{N}$ be the matrix

$$\mathcal{N} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & b_1 & \cdots & b_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & b_1^{\epsilon n-1} & \cdots & b_{n-1}^{\epsilon n-1} \end{pmatrix},$$

   where $1, b_1, \ldots, b_{n-1}$ are n different elements in $\mathbb{F}_{2^a}$. The servers locally compute

$$\begin{pmatrix} \langle \boldsymbol{r}_1^{V_j}\rangle \\ \langle \boldsymbol{r}_2^{V_j}\rangle \\ \vdots \\ \langle \boldsymbol{r}_{\epsilon n}^{V_j}\rangle \end{pmatrix} = \mathcal{N} \cdot \begin{pmatrix} \langle \boldsymbol{s}_1^{V_j}\rangle \\ \langle \boldsymbol{s}_2^{V_j}\rangle \\ \vdots \\ \langle \boldsymbol{s}_n^{V_j}\rangle \end{pmatrix},$$

where $\{\boldsymbol{r}_i^{V_j}\}_{j=1}^N$ form a $\Sigma_{\times a}^{(2)}$-sharing $[\boldsymbol{r}_i]_a^{(2)}$ for $i = 1, \ldots, \epsilon n$.

(b) The servers locally compute the additive sharing of V_j's shares of $[\boldsymbol{r}_i^{(1)}]^{(2)}, \ldots, [\boldsymbol{r}_i^{(a)}]^{(2)}$ from $\langle \boldsymbol{r}_i^{V_j} \rangle$ for each $i = 1, \ldots, \epsilon n$.

2. **Computing Zero Sharings.** We group the random sharings with all-zero secrets in Step 4 of the Sharing Phase (while emulating $\mathcal{F}_{\mathsf{prep}}$) into $10WN/(ank^2\epsilon) + W_O/(ank\epsilon)$ groups, where the servers receive $\langle \boldsymbol{s}_i^{V_j} \rangle$ for $i = 1, \ldots, n$ in each group (each $\langle \boldsymbol{s}_i^{V_j} \rangle$ is generated by server S_i). Then:

(a) The servers locally compute

$$\begin{pmatrix} \langle \boldsymbol{o}_1^{V_j} \rangle \\ \langle \boldsymbol{o}_2^{V_j} \rangle \\ \vdots \\ \langle \boldsymbol{o}_{\epsilon n}^{V_j} \rangle \end{pmatrix} = \mathcal{N} \cdot \begin{pmatrix} \langle \boldsymbol{s}_1^{V_j} \rangle \\ \langle \boldsymbol{s}_2^{V_j} \rangle \\ \vdots \\ \langle \boldsymbol{s}_n^{V_j} \rangle \end{pmatrix},$$

where $\{\boldsymbol{o}_i^{V_j}\}_{j=1}^n$ form a $\Sigma_{\times a}^{(2)}$-sharing $[\boldsymbol{o}_i]_a^{(2)}$ for $i = 1, \ldots, \epsilon n$.

(b) The servers locally compute the additive sharing of V_j's shares of $[\boldsymbol{o}_i^{(1)}]^{(2)}, \ldots, [\boldsymbol{o}_i^{(a)}]^{(2)}$ from $\langle \boldsymbol{o}_i^{V_j} \rangle$ for each $i = 1, \ldots, \epsilon n$.

3. **Computing Masks for Output Sharings.** For each batch of k output wires attached to client C_i in circuit C', the servers add their shares of the additive sharing (of the form $\langle \boldsymbol{o}^{V_j} \rangle$) of a random $\Sigma^{(2)}$-sharing with an all-zero secret computed in the last step to this sharing $[\boldsymbol{r}']^{(2)}$ and set the result to be $[\boldsymbol{r}]^{(2)}$.

4. **Preprocessing for Transpose Protocols.** The servers do the following $10W/k^2$ times in parallel:

(a) For each server $S_{i,\alpha}$, take one sharing generated by $S_{i,\alpha}$ in Step 5 of the Sharing Phase (while emulating $\mathcal{F}_{\mathsf{prep}}$, the same for the remaining in this step) where V_j's share is $\boldsymbol{r}_{i,\alpha}^{V_j}$. The servers compute their shares of $\langle \boldsymbol{r}_i^{V_j} \rangle = \sum_{\alpha=1}^c \langle \boldsymbol{r}_{i,\alpha}^{V_j} \rangle$ for $i = 1, \ldots, N$. Then, $\{\boldsymbol{r}_i^{V_\alpha}\}_{\alpha=1}^N$ form a $\Sigma^{(2)}$-sharing $[\boldsymbol{r}_i']^{(2)}$. Then the servers add their shares of the additive sharing (of the form $\langle \boldsymbol{o}^{V_\alpha} \rangle$) of a random $\Sigma^{(2)}$-sharing with an all-zero secret prepared in Step 4 of the Sharing Phase to this sharing $[\boldsymbol{r}_i']^{(2)}$ and set the result to be $[\boldsymbol{r}_i]^{(2)}$.

(b) The servers group $N\ell$ random $\Sigma^{(2)}$-sharings $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{N\ell}]^{(2)}$ prepared in Step 3 of the sharing phase together and associate them with an execution of $\Pi_{\mathsf{Transpose}}$. Here the shares of $[\boldsymbol{u}_1]^{(2)}, \ldots, [\boldsymbol{u}_{N\ell}]^{(2)}$ for each virtual server V_i are shared among $S_{i,1}, \ldots, S_{i,c}$ by additive sharings $\langle \boldsymbol{u}_1^{V_i} \rangle, \ldots, \langle \boldsymbol{u}_{N\ell}^{V_i} \rangle$.

Figure 30: The local computation phase of the dishonest majority protocol Π_2.

Garbling Phase. In the garbling phase, the real-world servers emulate the virtual servers to garble their local circuits by using the multiparty garbling technique. This phase only contains the local computation of each virtual server, which corresponds to Step 4 in Π_1'. Communication is only required between the servers that participate in the emulation of the same virtual server in this phase.

Protocol $\Pi_{2\text{-Garble}}$

Garbling Phase

Let $\mathcal{O}_1$ be a random oracle with output length κ and $\mathcal{O}_2$ be a random oracle with output length $c\ell\kappa$. For each virtual server V_j with local circuit $\mathtt{Circ}^{V_j}$ in Π_0, the servers $S_{j,1}, \ldots, S_{j,c}$ do the following:

1. **Computing Output Labels.**

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

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

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

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).

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

2. **Garbling Local Circuits.**

(a) For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, each server $S_{j,i}$ computes

$$k^{S_{j,i}}_{o,0} \| \lambda^{S_{j,i}}_o = (k^{S_{j,i}}_{a,0} \| \lambda^{S_{j,i}}_a) \oplus (k^{S_{j,i}}_{b,0} \| \lambda^{S_{j,i}}_b).$$

This computation is performed gate by gate.

(b) For each wire w in $\mathtt{Circ}^{V_j}$ that is not an output wire of an output gate. Each server $S_{j,i}$ computes $k^{S_{j,i}}_{w,1} = k^{S_{j,i}}_{w,0} \oplus \Delta^{S_{j,i}}$.

(c) For each AND gate g in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, the servers hold $\langle k^{S_{j,\alpha}}_{a,0}\rangle$ where $S_{j,\alpha}$'s share is $k^{S_{j,\alpha}}_{o,0}$ and all other servers have all-0 shares. Similarly, they hold $\langle k^{S_{j,\alpha}}_{o,1}\rangle$, and they also hold $\langle \lambda_a\rangle, \langle \lambda_b\rangle, \langle \lambda_o\rangle$. Then:

i. Each pair of servers takes one result from $\mathcal{F}_{\mathsf{ROT}}$ generated in the Sharing Phase. Then the servers run Π_{Mult} to compute $\langle \lambda_a \cdot \lambda_b\rangle$. Then the servers locally compute $\langle \chi_1\rangle, \langle \chi_2\rangle, \langle \chi_3\rangle, \langle \chi_4\rangle$, where

$$\chi_1 = ((0 \oplus \lambda_a) \wedge (0 \oplus \lambda_b)) \oplus \lambda_o, \qquad \chi_2 = ((0 \oplus \lambda_a) \wedge (1 \oplus \lambda_b)) \oplus \lambda_o,$$
$$\chi_3 = ((1 \oplus \lambda_a) \wedge (0 \oplus \lambda_b)) \oplus \lambda_o, \qquad \chi_4 = ((1 \oplus \lambda_a) \wedge (1 \oplus \lambda_b)) \oplus \lambda_o.$$

ii. Each pair of servers takes $4c$ results from $\mathcal{F}_{\mathsf{ROT}}$ generated in the Sharing Phase. Then the servers run Π_{Mult} to compute $\langle \chi_i \cdot (k^{S_{j,\alpha}}_{o,1} - k^{S_{j,\alpha}}_{o,0})\rangle$ for $i = 1, 2, 3, 4$ and $\alpha = 1, \ldots, c$. Then the servers locally compute $\langle k_{o,\chi_i}\rangle^{S_{j,\alpha}} = \langle k^{S_{j,\alpha}}_{o,0} \oplus \chi_i \cdot (k^{S_{j,\alpha}}_{o,1} - k^{S_{j,\alpha}}_{o,0})\rangle$ for $i = 1, 2, 3, 4$ and $\alpha = 1, \ldots, c$.

iii. Each server $S_{j,i}$ calls the random oracle $\mathcal{O}_1$ with input $k^{S_{j,i}}_{a,i_0} \| i_0 \| k^{S_{j,i}}_{b,i_1} \| i_1 \| i \| j \| \alpha \| g$ for each $(i_0, i_1) = (0,0), (0,1), (1,0), (1,1)$ and $\alpha = 1, \ldots, c$ and then receives the output. The output can be regarded as additively shared among $S_{j,1}, \ldots, S_{j,c}$, where all the servers except $S_{j,i}$ have all-0 shares.

iv. The servers locally compute $\langle A^{S_{j,\alpha}}_{g,2i_0+i_1}\rangle$ for each $(i_0, i_1) = (0,0), (0,1), (1,0), (1,1)$ and $\alpha = 1, \ldots, c$, where

$$A^{S_{j,\alpha}}_{g,2i_0+i_1} = \left(\bigoplus_{i=1}^{c} \left(\mathcal{O}_1(k^{S_{j,i}}_{a,i_0} \| i_0 \| k^{S_{j,i}}_{b,i_1} \| i_1 \| i \| j \| \alpha \| g) \right) \right) \oplus (k^{S_{j,\alpha}}_{o,\chi_{2i_0+i_1}} \| \chi_{2i_0+i_1}).$$

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

(d) For each output gate (with index $\mathsf{k} = (i-1)\ell + a$ that outputs the a-th bit of $\boldsymbol{s}_i$) of $\mathtt{Circ}^{V_j}$ with input wire w, the servers hold $\langle Y^{S_{j,\alpha}}_{\mathsf{k},0}\rangle$ where $S_{j,\alpha}$'s share is $Y^{S_{j,\alpha}}_{\mathsf{k},0}$ and all other servers have all-0 shares. Similarly, they hold $\langle Y^{S_{j,\alpha}}_{\mathsf{k},1}\rangle$, and they also hold $\langle \lambda_w\rangle$. Then:

i. Each pair of servers takes $2c\ell$ results from $\mathcal{F}_{\mathsf{ROT}}$ generated in the Sharing Phase. Then the servers run Π_{Mult} to compute $\langle (i_2 \oplus \lambda_w) \cdot (Y^{S_{j,\alpha}}_{\mathsf{k},1} - Y^{S_{j,\alpha}}_{\mathsf{k},0})\rangle$ for $i_2 = 0, 1$ and $\alpha = 1, \ldots, c$. Then the servers locally compute $\langle Y^{S_{j,\alpha}}_{\mathsf{k},i_2 \oplus \lambda_w}\rangle = \langle Y^{S_{j,\alpha}}_{\mathsf{k},0} \oplus (i_2 \oplus \lambda_w) \cdot (Y^{S_{j,\alpha}}_{\mathsf{k},1} - Y^{S_{j,\alpha}}_{\mathsf{k},0})\rangle$ for $i_2 = 0, 1$ and $\alpha = 1, \ldots, c$.

ii. Each server $S_{j,i}$ calls the random oracle $\mathcal{O}_2$ with input $k^{S_{j,\alpha}}_{w,i_2} \| i_2 \| i \| j \| \alpha \| w$ for each $i_2 = 0, 1$ and $\alpha = 1, \ldots, c$ and then receives the output. The output can be regarded as additively shared among $S_{j,1}, \ldots, S_{j,c}$, where all the servers except $S_{j,i}$ have all-0 shares.

iii. The servers locally compute $\langle \mathtt{ct}_{w,i_2}^{S_{j,\alpha}} \rangle$ for each $i_2 = 0, 1$ and $\alpha = 1, \ldots, c$, where

$$\mathtt{ct}_{w,i_2}^{S_{j,\alpha}} = \left(\bigoplus_{i=1}^{c} \left(\mathcal{O}_2(k_{w,i_2}^{S_{j,i}} \| i_2 \| i \| j \| \alpha \| w) \right) \right) \oplus Y_{\mathsf{k}, i_2 \oplus \lambda_w}^{S_{j,\alpha}},$$

Let $\mathbf{ct}_{w,i_2}^{V_j} = (\mathtt{ct}_{w,i_2}^{S_{j,1}}, \ldots, \mathtt{ct}_{w,i_2}^{S_{j,c}})$ for each $i_2 = 0, 1$. The servers then get $\langle \mathbf{ct}_{w,0}^{V_j} \rangle, \langle \mathbf{ct}_{w,1}^{V_j} \rangle$.

3. **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 $\Pi_{\mathsf{Open}}(\langle x_w \oplus \lambda_w \rangle)$.

Figure 31: The garbling phase of the dishonest majority protocol Π_2.

Verification Phase. In the verification phase, the servers emulate the virtual servers to do the verification process Π_{ver} in Π_{prep} of Π_0. Besides, each server chooses his watchlist and asks the servers that emulate each virtual server on his watchlist to open their commitments of the input and local randomness to the local computation of this virtual server. Then, each server checks whether the local computation of all the virtual servers on his watchlist is correctly performed.

Protocol $\Pi_{2\text{-Ver}}$

Verification Phase

1. **Verification of the Sharings.** The virtual servers run the verification process below to verify the $\Sigma^{(2)}$-sharings generated in Step 3 and Step 4 of the Sharing Phase, the random mask $\Sigma^{(2)}$-sharings for output sharings generated by the clients in Step 2 of the Sharing Phase, the random $\Sigma^{(2)}$-sharings generated by the servers in Step 5 of the Sharing Phase, and the input Σ-sharings. We denote the Σ-sharings to be checked by $[\boldsymbol{x}_1], \ldots, [\boldsymbol{x}_{k_1}]$, the $\Sigma^{(2)}$-sharings (excluding the sharings prepared by Π_{Zero}) by $[\boldsymbol{x}'_1]^{(2)}, \ldots, [\boldsymbol{x}'_{k_2}]^{(2)}$, and the $\Sigma^{(2)}$-sharings prepared by Π_{Zero} by $[\boldsymbol{o}_1]^{(2)}, \ldots, [\boldsymbol{o}_{k_3}]^{(2)}$, where each virtual server V_j's share is shared by an additive sharing among $S_{j,1}, \ldots, S_{j,c}$. Like in Π_{ver}, the servers view the sharings to be checked as $\Sigma_{\times\kappa'}$-sharings and $\Sigma^{(2)}_{\times\kappa'}$-sharings.
   (a) The servers invoke $\mathcal{F}_{\mathsf{Coin}}$ to get $s \in \mathbb{F}_{2^{\kappa'}}$. If abort is received, abort the protocol. Then the servers expand s to a vector $(s_1, \ldots, s_{k_1}, s'_1, \ldots, s'_{k_2}, s^{(1)}, \ldots, s^{(k_3)}) \in \mathbb{F}_{2^{\kappa'}}^{k_1+k_2+k_3}$ via a pseudorandom generator.
   (b) The servers $S_{\alpha,1}, \ldots, S_{\alpha,c}$ of each virtual server V_α locally compute an additive sharing of V_α's share of $[\tau]_{\kappa'} = \sum_{j=1}^{k_1} s_j \cdot [\boldsymbol{x}_j]_{\kappa'} + \sum_{i=1}^{n} [\boldsymbol{r}^{(i)}]_{\kappa'}$ by computing

$$\sum_{j=1}^{k_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i=1}^{n} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle,$$

   where $\boldsymbol{x}_j^{V_\alpha}, \boldsymbol{r}^{(i,V_\alpha)}$ are V_α's shares of $[\boldsymbol{x}_j]_{\kappa'}, [\boldsymbol{r}^{(i)}]_{\kappa'}$ respectively. Similarly, $S_{\alpha,1}, \ldots, S_{\alpha,c}$ compute an additive sharing of V_α's share of $[\tau']^{(2)}_{\kappa'} = \sum_{j=1}^{k_2} s'_j \cdot [\boldsymbol{x}'_j]^{(2)}_{\kappa'} + \sum_{i=1}^{n} [\boldsymbol{r}^{(i)'}]^{(2)}_{\kappa'}$ by computing

$$\sum_{j=1}^{k_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i=1}^{n} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle$$

   where $\boldsymbol{x}_j^{V'_\alpha}, \boldsymbol{r}^{(i,V_\alpha)'}$ are V_α's shares of $[\boldsymbol{x}'_j]^{(2)}_{\kappa'}, [\boldsymbol{r}^{(i)'}]^{(2)}_{\kappa'}$ respectively, and compute an additive sharing of V_α's share of $[\tau_0]^{(2)}_{\kappa'} = \sum_{j=1}^{k_3} s^{(j)} \cdot [\boldsymbol{o}_j]^{(2)}_{\kappa'} + \sum_{i=1}^{n} [\boldsymbol{o}^{(i)}]^{(2)}_{\kappa'}$ by computing

$$\sum_{j=1}^{k_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i=1}^{n} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle$$

   where $\boldsymbol{o}_j^{V_\alpha}, \boldsymbol{o}^{(i,V_\alpha)}$ are V_α's shares of $[\boldsymbol{o}_j]^{(2)}_{\kappa'}, [\boldsymbol{o}^{(i)}]^{(2)}_{\kappa'}$ respectively.

(c) The servers $S_{\alpha,1},\ldots,S_{\alpha,c}$ sends the additive sharings of V_α's shares of $[\tau]_{\kappa'},[\tau']^{(2)}_{\kappa'},[\tau_0]^{(2)}_{\kappa'}$ to all the servers.

(d) Each server reconstructs $[\tau]_{\kappa'},[\tau']^{(2)}_{\kappa'},[\tau_0]^{(2)}_{\kappa'}$ and sends $\mathcal{O}_1([\tau]_{\kappa'},[\tau']^{(2)}_{\kappa'},[\tau_0]^{(2)}_{\kappa'})$ to all the other servers. Then, all the servers check whether the results received from all the servers are the same. If not, abort the protocol.

(e) Each server checks whether $[\tau]_{\kappa'}$ is a valid $\Sigma_{\times\kappa'}$-sharing, whether $[\tau']^{(2)}_{\kappa'}$ is a valid $\Sigma^{(2)}_{\times\kappa'}$-sharing, and whether $[\tau_0]^{(2)}_{\kappa'}$ is a valid $\Sigma^{(2)}_{\times\kappa'}$-sharing with an all-zero secret. If not, abort the protocol.

2. **Verification of Local Computation.**

(a) Each server S sends a set $\mathsf{Ver}^S \subset \{V_1,\ldots,V_N\}$ of size $N/16n$ to all the servers. Then all the servers send this set to each other to check that all the servers receive the same set. Then for each virtual server $V_j \in \mathsf{Ver}^S$, each $S_{j,\alpha}$ sends $(\mathsf{open}, S_{j,\alpha}, \tau_{\mathsf{IS}^{S_{j,\alpha}}}, S)$ to $\mathcal{F}_{\mathsf{Commit}}$ and sends all the messages he sends and receives in the garbling phase and Step 1 of the verification phase to S. Then, each party P sends all the shares he generates for $S_{j,1},\ldots,S_{j,c}$ in the sharing phase to S for each virtual server $V_j \in \mathsf{Ver}^S$.

(b) Each server S checks whether $S_{j,1},\ldots,S_{j,c}$ perform the computation of V_j correctly for each $V_j \in \mathsf{Ver}^S$, whether the additive sharings of V_j's shares opened by the generator match the values committed by the servers, and whether for each pair of servers $(S_{j,\alpha},S_{j,\beta})$, the committed output from $\mathcal{F}_{\mathsf{ROT}}$ is valid, i.e. $r^{(\alpha,\beta)}_{b^{(\alpha,\beta)}}$ committed by $S_{j,\beta}$ is among $r^{(\alpha,\beta)}_0, r^{(\alpha,\beta)}_1$ committed by $S_{j,\alpha}$.

Figure 32: The verification phase of the dishonest majority protocol Π_2.

Evaluation Phase. In the evaluation phase, the servers send their shares of the output masks, the input labels, and the garbled circuits to P_{king} to let P_{king} evaluate each virtual server's garbled circuit and send the output to each client. This corresponds to Steps 5-8 of Π'_1.

Protocol $\Pi_{2\text{-Eval}}$

Evaluation Phase

1. **Sending Output Masks.** For each input wire w of an output gate in local circuit $\mathtt{Circ}^{V_j}$, the servers $S_{j,1},\ldots,S_{j,c}$ send their shares of $\langle\lambda_w\rangle$ to P_{king}. P_{king} then reconstructs λ_w.

2. **Encrypting Input Labels.** For each $i=1,\ldots,\mathsf{rec}$, if R_i is the virtual server V_j and 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 server $S_{j,\beta}$ queries the random oracle $\mathcal{O}_1$ 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 $\boldsymbol{r}_{b,\eta,\beta}=(r^{(1)}_{b,\eta,\beta},\ldots,r^{(\kappa)}_{b,\eta,\beta})$ for each $b=0,1$ with each $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{b},\beta}=(r^{(\alpha)}_{b,1,\beta},\ldots,r^{(\alpha)}_{b,k,\beta})$. Then $S_{j,\beta}$ receives $\mathcal{O}_1(\boldsymbol{r}_{0,\eta,\beta}\|0\|i\|\beta\|\eta\|j_\eta)$ and $\mathcal{O}_1(\boldsymbol{r}_{1,\eta,\beta}\|1\|i\|\beta\|\eta\|j_\eta)$.

(b) Each server $S_{j,\beta}$ locally computes

$$\mathsf{ct}^{(i,\beta)}_{j_\eta,0}=\mathcal{O}_1(\boldsymbol{r}_{0,\eta,\beta}\|0\|i\|\beta\|\eta\|j_\eta)\oplus\left(k^{S_{j,\beta}}_{w_{j_\eta},\lambda_{w_{j_\eta}}}\|\lambda_{w_{j_\eta}}\right)$$

and

$$\mathsf{ct}^{(i,\beta)}_{j_\eta,1}=\mathcal{O}_1(\boldsymbol{r}_{1,\eta,\beta}\|1\|i\|\beta\|\eta\|j_\eta)\oplus\left(k^{S_{j,\beta}}_{w_{j_\eta},1\oplus\lambda_{w_{j_\eta}}}\|(1\oplus\lambda_{w_{j_\eta}})\right).$$

Then $S_{j,\beta}$ sends the ciphertexts $\mathsf{ct}^{(i,\beta)}_{j_\eta,0},\mathsf{ct}^{(i,\beta)}_{j_\eta,1}$ to P_{king}.

3. **Sending Input Labels.** For each input wire w of each virtual server V_j's local circuit $\mathtt{Circ}^{V_j}$, if the wire value x_w doesn't come from a reconstruction of a $\Sigma^{(2)}$-sharing:

(a) The servers $S_{j,1},\ldots,S_{j,c}$ send $x_w\oplus\lambda_w$ to P_{king}. P_{king} checks whether the servers among $S_{j,1},\ldots,S_{j,c}$ send the same value. If not, P_{king} aborts the protocol.

(b) Each server $S_{j,i}$ sends $k^{S_{j,i}}_{w,x_w\oplus\lambda_w}$ to P_{king}.

4. **Sending Garbled Circuits.** For each $j=1,\ldots,N$, each server $S_{j,\beta}$ sends his shares of $\langle\boldsymbol{A}^{V_j}_{g,1}\rangle,\langle\boldsymbol{A}^{V_j}_{g,2}\rangle,\langle\boldsymbol{A}^{V_j}_{g,3}\rangle,\langle\boldsymbol{A}^{V_j}_{g,4}\rangle$ for each AND gate and $\langle\mathbf{ct}^{V_j}_{w,0}\rangle,\langle\mathbf{ct}^{V_j}_{w,1}\rangle$ for each output wire w of the circuits

to P_{king}. P_{king} then reconstructs the secrets. For each $\beta = 1, \ldots, c$, P_{king} obtains $A_{g,1}^{S_{j,\beta}}, A_{g,2}^{S_{j,\beta}}, A_{g,3}^{S_{j,\beta}}, A_{g,4}^{S_{j,\beta}}$ for each AND gate and $\mathsf{ct}_{w,0}^{S_{j,\beta}}, \mathsf{ct}_{w,1}^{S_{j,\beta}}$ for each output wire w.

5. **Evaluating the Circuit.** The evaluator P_{king} evaluates the circuit by doing the following:

   (a) For $j = 1, \ldots, N$, since no input to $\mathtt{Circ}_1^{V_j}$ comes from reconstruction, P_{king} already gets the input labels $k_{w,x_w\oplus\lambda_w}^{S_{j,1}}, \ldots, k_{w,x_w\oplus\lambda_w}^{S_{j,c}}$ 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_{king} computes $k_{o,x_o\oplus\lambda_o}^{S_{j,\beta}} = k_{a,x_a\oplus\lambda_a}^{S_{j,\beta}} \oplus k_{b,x_b\oplus\lambda_b}^{S_{j,\beta}}$ for each $\beta = 1, \ldots, c$ 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_{king} computes

$$k_{o,x_o\oplus\lambda_o}^{S_{j,\beta}} \| (x_o \oplus \lambda_o) = A_{g,2(x_a\oplus\lambda_a)+(x_b\oplus\lambda_b)}^{S_{j,\beta}} \oplus \left(\bigoplus_{i=1}^{c} \left(\mathcal{O}_1(k_{a,x_a\oplus\lambda_a}^{S_{j,i}} \| (x_a \oplus \lambda_a) \| k_{b,x_b\oplus\lambda_b}^{S_{j,i}} \| (x_b \oplus \lambda_b) \| i \| j \| \beta \| g) \right) \right)$$

     for each $\beta = 1, \ldots, c$.
   - If g is an output gate (indexed k) with output wire w, P_{king} computes

$$Y_{\mathsf{k},x_w}^{S_{j,\beta}} = \mathsf{ct}_{w,x_w\oplus\lambda_w}^{S_{j,\beta}} \oplus \left(\bigoplus_{i=1}^{c} \mathcal{O}_2(k_{w,x_w\oplus\lambda_w}^{S_{j,i}} \| (x_w \oplus \lambda_w) \| i \| j \| \beta \| w) \right)$$

     for each $\beta = 1, \ldots, c$.

   (b) After evaluating all the gate of $\mathtt{Circ}_1^{V_j}$ for $j = 1, \ldots, N$, P_{king} obtains the sharings $[\boldsymbol{s}_1]^{(2)}$ and $\{Y_{a,s_{1,a}^{V_j}}^{S_{j,\beta}}\}_{a=1}^{\ell^2}$ for each $\beta = 1, \ldots, c$, where $s_{1,a}^{V_j}$ is the a-th bit of V_j's share of $[\boldsymbol{s}_1]^{(2)}$. Then P_{king} checks whether the sharings $[\boldsymbol{s}_1]^{(2)}$ and $\{[\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}]^{(3)}\}_{\alpha=1}^{\kappa}$ for each $\beta = 1, \ldots, c$ are all valid. If not, P_{king} aborts the protocol. Otherwise, P_{king} reconstructs $\boldsymbol{s}_1$ and $\{\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}\}_{\alpha=1}^{\kappa}$ for each $\beta = 1, \ldots, c$.

   (c) For each $i = 1, \ldots, \mathsf{rec}$, if R_1 is a virtual server V_j and the η-th bit $s_{1,\eta}$ of $\boldsymbol{s}_1$ is used as an input wire with index j_η in R_1's circuit $\mathtt{Circ}^{V_j}$, with $\boldsymbol{s}_1$ and $\{\boldsymbol{r}_{\boldsymbol{s}_1,\beta}^{(\alpha)}\}_{\alpha=1}^{\kappa}$, P_{king} decrypts $k_{w_{j_\eta},s_{1,\eta}\oplus\lambda_{w_{j_\eta}}}^{S_{j,\beta}} \| (s_{1,\eta} \oplus \lambda_{w_{j_\eta}})$ from the ciphertexts $\mathsf{ct}_{j_\eta,s_{1,\eta}}^{(i,\beta)}$ for each $\beta = 1, \ldots, c$ by

$$k_{w_{j_\eta},s_{1,\eta}\oplus\lambda_{w_{j_\eta}}}^{S_{j,\beta}} \| (s_{1,\eta} \oplus \lambda_{w_{j_\eta}}) = \mathsf{ct}_{j_\eta,s_{1,\eta}}^{(1,\beta)} \oplus \mathcal{O}_1\big(\boldsymbol{r}_{s_{1,\eta},\eta,\beta} \| s_{1,\eta} \| 1 \| \beta \| \eta \| j_\eta\big).$$

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

6. **Sending Outputs.**

   (a) For each client receiver R_i, P_{king} sends $\boldsymbol{s}_i$ and $\{\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}\}_{\alpha=1}^{\kappa}$ for each $\beta = 1, \ldots, c$ to R_i.

   (b) Each client receiver R_i checks whether $\{\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}\}_{\alpha=1}^{\kappa}$ matches $\boldsymbol{s}_i$ and $(\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}, \boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)})$ for each $\beta = 1, \ldots, c$ generated in the garbling phase. If not, R_i aborts the protocol. Otherwise, R_i computes $\mathtt{Circ}^{R_i}$ to get his output locally. Then each client computes $\mathtt{Circ}^{R_i}$ to get his output locally.

Figure 33: The evaluation phase of the dishonest majority protocol Π_2.

Theorem 10. *Let Σ be an (N, t, k, ℓ)-LSSS over $\mathbb{F}_2$ with 3-multiplicative reconstruction and Π_0 be a protocol that has the properties listed in Section 4.1. Protocol Π_2 securely realizes $\mathcal{F}$ in the $\{\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Commit}}, \mathcal{F}_{\mathsf{ROT}}\}$-hybrid model against a fully malicious adversary that corrupts any number of clients and at most $(1-\epsilon)n$ servers.*

J.3 Security Proof

Proof. We prove the security of Π_2 by constructing an ideal adversary Sim_2. Then we will show that the output in the ideal world is computationally indistinguishable from that in the real world using hybrid arguments. Our simulation is in the client-server model where the adversary corrupts any number of clients and exactly $(1-\epsilon)n$ servers.

Without loss of generality, we suppose that P_{king} is corrupted. We give the ideal adversary Sim_2 below.

Simulator $\mathsf{Sim}_{2\text{-}\mathsf{Share}}$

Sharing Phase

Determining the Virtual Servers:

1. Sim_2 emulates $\mathcal{F}_{\mathsf{Coin}}$ to receive $\mathsf{RandCoin}$ from all the corrupted parties and follows the protocol to sample $r \in \mathbb{F}_{2^{\kappa'}}$ randomly. Then Sim_2 emulates $\mathcal{F}_{\mathsf{Coin}}$ to send r to $\mathcal{A}$. If abort is received from $\mathcal{A}$, Sim_2 emulates $\mathcal{F}_{\mathsf{Coin}}$ to send abort to all the corrupted parties and aborts the protocol. Then, Sim_2 follows the protocol to determine the virtual servers and receive the result of $\{S_{j,1}, \ldots, S_{j,c}\}$ for each $j = 1, \ldots, N$. If there are more than $N/16$ corrupted virtual servers, Sim_2 aborts the simulation. Without loss of generality, for each honest virtual server V_j, we assume that only $S_{j,1}$ is honest. In the case that $S_{j,i}$ for some $i > 1$ is also honest, Sim_2 honestly follows the protocol for $S_{j,i}$.
2. Sim_2 sets $\mathsf{CompCheck} = \mathsf{CorrCheck} = \mathsf{OTCheck} = 0$.
3. For each honest virtual server V_j, Sim_2 sets $\mathsf{Corr}_j = \mathsf{Comp}_j = \mathsf{Check}_j = \mathsf{ROT}_j = 0$.

Emulating $\mathcal{F}_{\mathsf{prep}}$:

1. **Transforming the Circuit.** Sim_2 follows the protocol to transform the circuit C to C'.
2. **Preparing Masks for Output Sharings.** For each batch of k output wires attached to an honest client C_i in circuit C', Sim_2 randomly samples corrupted servers' shares of the additive sharing $\langle \boldsymbol{r}^{V_j} \rangle$ of each virtual server V_j's share $\boldsymbol{r}^{V_j}$ of $[\boldsymbol{r}']^{(2)}$ (where corrupted virtual servers' shares are sampled using the algorithm Alg_3 in Remark 1, same below) and sends them to the corrupted servers. For each batch of k input wires attached to a corrupted client C_i in circuit C', Sim_2 receives the honest server $S_{j,1}$'s share of the additive sharing of each honest virtual server V_j's shares of $[\boldsymbol{r}']^{(2)}$ from C_i.
3. **Preparing Random Sharings.** For each $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{s}]^{(2)}_a$ shared by an honest server S, Sim_2 randomly samples corrupted servers' shares of $\langle \boldsymbol{s}^{V_j} \rangle$ and sends them to the corrupted servers. For each $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{s}]^{(2)}_a$ shared by a corrupted server S, Sim_2 receives the honest server $S_{j,1}$'s share of the additive sharing of each honest virtual server V_j's share of $[\boldsymbol{s}]^{(2)}_a$ from S.
4. **Preparing Zero Sharings.** For each $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{s}]^{(2)}_a$ with an all-zero secret shared by an honest server S, Sim_2 randomly samples corrupted servers' shares of $\langle \boldsymbol{s}^{V_j} \rangle$ and sends them to the corrupted servers. For each $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{s}]^{(2)}_a$ shared by a corrupted server S, Sim_2 receives the honest server $S_{j,1}$'s share of the additive sharing of each honest virtual server V_j's share of $[\boldsymbol{s}]^{(2)}_a$ from S.
5. **Preparing Masks for Transpose Protocols.** For each $\Sigma^{(2)}$-sharing $[\boldsymbol{r}_{i,1}]^{(2)}_a$ shared by an honest server $S_{i,1}$, Sim_2 randomly samples corrupted servers' shares of $\langle \boldsymbol{r}^{V_\alpha}_{i,j} \rangle$ and sends them to the corrupted servers. For each $\Sigma^{(2)}$-sharing $[\boldsymbol{r}_{i,j}]^{(2)}_a$ shared by a corrupted server $S_{i,j}$, Sim_2 receives the honest server $S_{\alpha,1}$'s share of the additive sharing of each honest virtual server V_α's share of $[\boldsymbol{r}_{i,j}]^{(2)}_a$ from S.
6. **Preprocessing for the Verification of Sharings.** For each honest real-world server S_i, Sim_2 randomly samples corrupted servers' shares of the additive sharing of each virtual server's shares of $[\boldsymbol{r}^{(i)}]_{\kappa'}, [\boldsymbol{r}^{(i)'}]^{(2)}_{\kappa'}, [\boldsymbol{o}^{(i)}]^{(2)}_{\kappa'}$ and sends them to the corrupted servers. For each corrupted real-world server S_i, Sim_2 receives the honest server $S_{\alpha,1}$'s share of the additive sharing of each honest virtual server V_α's shares of $[\boldsymbol{r}^{(i)}]_{\kappa'}, [\boldsymbol{r}^{(i)'}]^{(2)}_{\kappa'}, [\boldsymbol{o}^{(i)}]^{(2)}_{\kappa'}$ from S_i.

Emulating $\mathcal{F}_{\mathsf{input}}$: For each batch of k input wires attached to an honest client C_i in circuit C', Sim_2 randomly samples corrupted servers' shares of the additive sharing of each virtual server's shares of $[\boldsymbol{s}]$ and sends them to the corrupted servers. For each batch of k input wires attached to a corrupted client C_i in

circuit C', Sim_2 receives the honest server $S_{j,1}$'s share of the additive sharing of each honest virtual server V_j's shares of $[\boldsymbol{s}]$ from C_i.

Preparing for the Garbling of Local Circuits:

1. **Calling Random OT.** For each honest virtual server V_j, for each invocation of $\mathcal{F}_{\mathsf{ROT}}$ of each pair of servers $(S_{j,1}, S_{j,\beta})$, Sim_2 receives $(b^{(1,\beta)}, r_{b^{(1,\beta)}}^{(1,\beta)})$ from $S_{j,\beta}$ and emulates $\mathcal{F}_{\mathsf{ROT}}$ to sends $(b^{(1,\beta)}, r_{b^{(1,\beta)}}^{(1,\beta)})$ to $S_{j,\beta}$. Similarly, for each invocation of $\mathcal{F}_{\mathsf{ROT}}$ of each pair of servers $(S_{j,\alpha}, S_{j,1})$, Sim_2 receives $(r_0^{(\alpha,1)}, r_1^{(\alpha,1)})$ from $S_{j,\alpha}$ and emulates $\mathcal{F}_{\mathsf{ROT}}$ to sends $(r_0^{(\alpha,1)}, r_1^{(\alpha,1)})$ to $S_{j,\alpha}$.
2. **Preparing for the Output Labels.** For each $i = 1, \ldots, \mathsf{rec}$:
   - If the receiver R_i of the i-th reconstruction of Π_0 is a corrupted virtual server or a corrupted client, for each honest virtual server V_j, Sim_2 receives $S_{j,1}$'s shares of the additive sharings of V_j's shares of $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$ for each $\beta = 1, \ldots, c$.
   - If the receiver R_i of the i-th reconstruction of Π_0 is an honest client, Sim_2 randomly samples corrupted servers' shares of each virtual server's shares of $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$ for $\beta = 1, \ldots, c$ and sends them to the corrupted servers on behalf of R_i.
   - If the receiver R_i of the i-th reconstruction of Π_0 is an honest virtual server V_j, Sim_2 randomly samples corrupted servers' shares of each virtual server's shares of $[\boldsymbol{r}_{\boldsymbol{0},1}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\boldsymbol{1},1}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},1}^{(\alpha)}]$ and sends them to the corrupted servers on behalf of $S_{j,1}$. Then, for each honest virtual server V_η, Sim_2 receives $S_{\eta,1}$'s shares of the additive sharings of V_η's shares of $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$ for each $\beta = 2, \ldots, c$.

Committing Local Input: Sim_2 emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{commit}, S_{j,i}, \mathsf{IS}^{S_{j,i}}, \tau_{\mathsf{IS}^{S_{j,i}}})$ from each corrupted server $S_{j,i}$ and emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(S_{j,i}, \tau_{\mathsf{IS}^{S_{j,i}}})$ to each corrupted server. Then:

1. For each $\Sigma, \Sigma^{(2)}$-sharing generated in this phase, Sim_2 receives all the shares of corrupted servers (of the additive sharings of virtual parties' shares) from the sets they committed.
2. For each sharing generated by an honest party, for each honest virtual server V_j, let the generated share of V_j be s, and then Sim_2 checks whether the committed inputs of $S_{j,2}, \ldots, S_{j,c}$'s shares of $\langle s \rangle$ matches what Sim_2 sends to them. If not, Sim_2 sets $\mathsf{Corr}_j = 1$.
3. For the results of $\mathcal{F}_{\mathsf{ROT}}$, Sim_2 checks whether for each honest virtual server V_j, the corrupted server $S_{j,2}, \ldots, S_{j,c}$ all committed their outputs correctly. For each honest virtual server V_j that fails in the check, Sim_2 sets $\mathsf{ROT}_j = 1$.
4. For each corrupted server $S_{j,i}$, Sim_2 retrieves $\lambda_w^{S_{j,i}}, k_{w,0}^{S_{j,i}}$ for each wire w that is not an output wire of an XOR gate of $\mathtt{Circ}^{V_j}$ and $\Delta^{S_{j,i}}$ from the commitment sent by $S_{j,i}$.
5. If for at least $N/12$ honest virtual server V_j it holds that $\mathsf{ROT}_j = 1$, Sim_2 sets $\mathsf{OTCheck} = 1$ and takes the first $N/12$ of them, and for each V_j of these taken virtual servers, Sim_2 randomly samples $S_{j,1}$'s share of each sharing generated by an honest party based on the corrupted servers' shares (using the algorithm Alg_4 in Remark 1, same below), samples the outputs of $\mathcal{F}_{\mathsf{ROT}}$ to $S_{j,1}$ based on the output to corrupted servers, and follows the protocol to sample $\lambda_w^{S_{j,1}}, k_{w,0}^{S_{j,1}}$ for each wire w that is not an output wire of an XOR gate or output gate in $\mathtt{Circ}^{V_j}$ and $\Delta^{S_{j,1}}$. For each of the other honest virtual servers V_j with $\mathsf{ROT}_j = 1$, Sim_2 sets $\mathsf{ROT}_j = 0$.
6. If there are over $N/48$ honest virtual servers V_j with $\mathsf{Corr}_j = 1$, Sim_2 sets $\mathsf{CorrCheck} = 1$.

Figure 34: The simulator for the sharing phase of Π_2 when P_{king} is corrupted.

Simulator $\mathsf{Sim}_{2\text{-Local}}$

Local Computation Phase

For each corrupted server $S_{j,i}$, Sim_2 follows the protocol to do all the local computation of $S_{j,i}$ with their committed inputs.

Figure 35: The simulator for the local computing phase of Π_2 when P_{king} is corrupted.

Simulator $\mathsf{Sim}_{2\text{-Garble}}$

Garbling Phase

For each honest virtual server V_j with $\mathsf{ROT}_j = 0$:

1. **Computing Output Labels.** For each $i = 2, \ldots, c$, Sim_2 follows the protocol to compute $Y_{a,b}^{S_{j,i}}$ for each $a = 1, \ldots, \ell^2\mathsf{rec}$ and $b = 0, 1$ with their committed inputs.
2. **Garbling Local Circuits.**
   (a) For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wires a, b and output wire o, Sim_2 follows the protocol to compute
   $$k_{o,0}^{S_{j,i}} \| \lambda_o^{S_{j,i}} = (k_{a,0}^{S_{j,i}} \| \lambda_a^{S_{j,i}}) \oplus (k_{b,0}^{S_{j,i}} \| \lambda_b^{S_{j,i}})$$
   gate by gate for each $i = 2, \ldots, c$.
   (b) For each wire w in $\mathtt{Circ}^{V_j}$ that is not an output wire of an output gate. Sim_2 follows the protocol to compute $k_{w,1}^{S_{j,i}} = k_{w,0}^{S_{j,i}} \oplus \Delta^{S_{j,i}}$ for each $i = 2, \ldots, c$.
   (c) For each AND gate g in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o:
      i. For each execution of Π_{Mult} in this step, for each server $S_{j,i}$ where $i = 2, \ldots, c$, Sim_2 samples a random κ-bit string as $r_1^{(1,i)} \oplus r_0^{(1,i)} \oplus x^{(1)}$, samples a random bit as $y^{(1)} \oplus b^{(i,1)}$, and sends them to $S_{j,i}$ on behalf of $S_{j,1}$. Then Sim_2 receives $r_1^{(i,1)} \oplus r_0^{(i,1)} \oplus x^{(i)}$ and $y^{(i)} \oplus b^{(1,i)}$ from $S_{j,i}$ for each $i = 2, \ldots, c$ and checks whether they are correctly computed with their committed inputs. If not, Sim_2 sets $\mathsf{Comp}_j = 1$.
      ii. Sim_2 honestly emulates the random oracle $\mathcal{O}_1$ and compute $S_{j,2}, \ldots, S_{j,c}$'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 $S_{j,1}$, and the outputs of $\mathcal{O}_1$.
   (d) For each output gate (with index $\mathsf{k} = (i-1)\ell + a$ that outputs the a-th bit of $\boldsymbol{s}_i$) of $\mathtt{Circ}^{V_j}$ with input wire w:
      i. For each execution of Π_{Mult} in this step, for each server $S_{j,i}$ where $i = 2, \ldots, c$, Sim_2 samples a random $(\kappa - 1)$-bit string as $r_1^{(1,i)} \oplus r_0^{(1,i)} \oplus x^{(1)}$, samples a random bit as $y^{(1)} \oplus b^{(i,1)}$, and sends them to $S_{j,i}$ on behalf of $S_{j,1}$. Then Sim_2 receives $r_1^{(i,1)} \oplus r_0^{(i,1)} \oplus x^{(i)}$ and $y^{(i)} \oplus b^{(1,i)}$ from $S_{j,i}$ for each $i = 2, \ldots, c$ and checks whether they are correctly computed with their committed inputs. If not, Sim_2 sets $\mathsf{Comp}_j = 1$.
      ii. Sim_2 honestly emulates the random oracle $\mathcal{O}_2$ and computes $S_{j,2}, \ldots, S_{j,c}$'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 $S_{j,1}$, and the outputs of $\mathcal{O}_2$.
3. **Masking Input Wire Values.** For each input wire w of $\mathtt{Circ}^{V_j}$:
   (a) Sim_2 samples a random bit as $S_{j,1}$'s share of $\langle x_w \oplus \lambda_w \rangle$ and sends it to $S_{j,2}, \ldots, S_{j,c}$ on behalf of $S_{j,1}$.
   (b) Sim_2 receives $S_{j,2}, \ldots, S_{j,c}$'s shares of $\langle x_w \oplus \lambda_w \rangle$ and checks whether they match the inputs they committed. If not, Sim_2 sets $\mathsf{Comp}_j = 1$. Otherwise, Sim_2 reconstructs $x_w \oplus \lambda_w$.

For each honest virtual server V_j with $\mathsf{ROT}_j = 1$, Sim_2 honestly emulates $S_{j,1}$ and the random oracles to communicate with $S_{j,2}, \ldots, S_{j,c}$ and follows the protocol to compute $S_{j,1}$'s shares of $\langle \boldsymbol{A}_{g,1}^{V_j} \rangle, \langle \boldsymbol{A}_{g,2}^{V_j} \rangle, \langle \boldsymbol{A}_{g,3}^{V_j} \rangle, \langle \boldsymbol{A}_{g,4}^{V_j} \rangle$ and $\langle \mathbf{ct}_{w,0}^{V_j} \rangle, \langle \mathbf{ct}_{w,1}^{V_j} \rangle$ based on the messages received from the corrupted servers. In addition, Sim_2 follows the protocol to compute all the messages that should be sent from a corrupted server with their committed inputs.

For each corrupted virtual server V_j, Sim_2 honestly emulate the random oracles to send outputs to $S_{j,1}, \ldots, S_{j,c}$.

Figure 36: The simulator for the garbling phase of Π_2 when P_{king} is corrupted.

Simulator $\mathsf{Sim}_{2\text{-Ver}}$

Verification Phase

1. **Verification of the Sharings.**

(a) Sim_2 emulates $\mathcal{F}_{\mathsf{Coin}}$ to receive $\mathsf{RandCoin}$ from all the corrupted servers and follows the protocol to sample $s \in \mathbb{F}_{2^{\kappa'}}$ randomly.

(b) Sim_2 emulates $\mathcal{F}_{\mathsf{Coin}}$ to send s to $\mathcal{A}$. If abort is received from $\mathcal{A}$, Sim_2 emulates $\mathcal{F}_{\mathsf{Coin}}$ to send abort to all the corrupted servers and aborts the protocol. Let the pseudorandom coefficients for the $\Sigma^{(2)}$-sharings expanded from s be $(s_1, \ldots, s_{k_1}, s'_1, \ldots, s'_{k_2}, s^{(1)}, \ldots, s^{(k_3)}) \in \mathbb{F}_{2^{\kappa'}}^{k_1+k_2+k_3}$.

(c) Let the set of the indices of Σ-sharings, random $\Sigma^{(2)}$-sharings, and random $\Sigma^{(2)}$-sharings with all-zero secrets (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 corrupted parties, Sim_2 computes the share s of each honest virtual server V_α based on $S_{\alpha,1}$'s share of $\langle s \rangle$ and the committed shares of $\langle s \rangle$ by $S_{\alpha,2}, \ldots, S_{\alpha,c}$. Then, Sim_2 computes $S_{\alpha,1}, \ldots, S_{\alpha,c}$'s shares of

$$\sum_{j \in \mathcal{C}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle,$$

$$\sum_{j \in \mathcal{C}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle,$$

and

$$\sum_{j \in \mathcal{C}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle.$$

(d) For each corrupted virtual server V_α and each honest virtual server V_α with $\mathsf{ROT}_\alpha = 1$, Sim_2 follows the protocol to compute $S_{\alpha,1}, \ldots, S_{\alpha,c}$'s shares of

$$\sum_{j \in \mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle,$$

$$\sum_{j \in \mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle,$$

and

$$\sum_{j \in \mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle.$$

Also for each honest virtual server V_α with $\mathsf{ROT}_\alpha = 0$, Sim_2 follows the protocol to compute $S_{\alpha,2}, \ldots, S_{\alpha,c}$'s shares (with the shares Sim_2 sends to them on behalf of honest parties in the sharing phase). Then, Sim_2 randomly samples the secret of the sharing for each honest virtual server V_α based on the fact that all the virtual servers' secrets form a $\Sigma_{\kappa'}$-sharing, a $\Sigma^{(2)}_{\kappa'}$-sharing, and a $\Sigma^{(2)}_{\kappa'}$-sharing with an all-zero secret. Then, for each honest virtual server V_α, Sim_2 computes $S_{\alpha,1}$'s shares based on the secrets and $S_{\alpha,2}, \ldots, S_{\alpha,c}$'s shares.

(e) For each honest virtual server V_α, Sim_2 computes $S_{\alpha,1}, \ldots, S_{\alpha,c}$'s shares of

$$\sum_{j=1}^{k_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i=1}^{n} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle = \sum_{j \in \mathcal{C}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle + \sum_{j \in \mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle,$$

$$\sum_{j=1}^{k_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i=1}^{n} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle = \sum_{j \in \mathcal{C}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle + \sum_{j \in \mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle,$$

and

$$\sum_{j=1}^{k_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i=1}^{n} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle = \sum_{j \in \mathcal{C}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle + \sum_{j \in \mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle.$$

For each honest virtual server V_α, Sim_2 then sends $S_{\alpha,1}$'s shares of the above sharings to all the servers on behalf of $S_{\alpha,1}$.

(f) Sim_2 receives all the corrupted servers' shares of additive sharings of each virtual server's shares of $[\tau]_{\kappa'}, [\tau']^{(2)}_{\kappa'}, [\tau_0]^{(2)}_{\kappa'}$. For each honest virtual server V_α with $\mathsf{ROT}_\alpha = 0$, Sim_2 checks whether the received corrupted servers' shares of the additive sharing of V_α's shares of $[\tau]_{\kappa'}, [\tau']^{(2)}_{\kappa'}, [\tau_0]^{(2)}_{\kappa'}$ match their shares computed from their committed inputs. If not, Sim_2 sets $\mathsf{Comp}_\alpha = 1$.

(g) For each honest server S, Sim_2 emulates each honest server to compute $[\tau]_{\kappa'}, [\tau']^{(2)}_{\kappa'}, [\tau_0]^{(2)}_{\kappa'}$ from what S receives in the last step and honestly emulates $\mathcal{O}_1$ to compute $\mathcal{O}_1([\tau]_{\kappa'}, [\tau']^{(2)}_{\kappa'}, [\tau_0]^{(2)}_{\kappa'})$ and send it to all the corrupted servers. Sim_2 honestly emulates $\mathcal{O}_1$ to interact with corrupted parties. Then, Sim_2 emulates each honest server S to receive $\mathcal{O}_1([\tau]_{\kappa'}, [\tau']^{(2)}_{\kappa'}, [\tau_0]^{(2)}_{\kappa'})$ from each corrupted server and check whether the sharings received from the corrupted servers are all the same as results $\mathcal{O}_1([\tau]_{\kappa'}, [\tau']^{(2)}_{\kappa'}, [\tau_0]^{(2)}_{\kappa'})$ generated by himself on behalf of the honest servers. If not, Sim_2 aborts the protocol on behalf of S. After completing the simulation, Sim_2 outputs what $\mathcal{A}$ outputs. If for two different honest servers, the sharings $[\tau]_{\kappa'}, [\tau']^{(2)}_{\kappa'}, [\tau_0]^{(2)}_{\kappa'}$ computed on behalf of them are different but $\mathcal{O}_1([\tau]_{\kappa'}, [\tau']^{(2)}_{\kappa'}, [\tau_0]^{(2)}_{\kappa'})$ are the same, Sim_2 aborts the simulation.

(h) Sim_2 follows the protocol to check whether $[\tau]_{\kappa'}$ is a valid $\Sigma_{\times\kappa'}$-sharing, whether $[\tau']^{(2)}_{\kappa'}$ is a valid $\Sigma^{(2)}_{\times\kappa'}$-sharing, and whether $[\tau_0]^{(2)}_{\kappa'}$ is a valid $\Sigma^{(2)}_{\times\kappa'}$-sharing with an all-zero secret on behalf of each honest server. If not, Sim_2 aborts the protocol on behalf of the honest server. After completing the simulation, Sim_2 outputs what $\mathcal{A}$ outputs.

(i) If there are over $N/48$ honest virtual servers V_j with $\mathsf{Comp}_j = 1$, Sim_2 sets $\mathsf{CompCheck} = 1$.

2. **Verification of Local Computation.**

(a) For each honest server S, Sim_2 follows the protocol to sample a set of $N/16n$ virtual servers as Ver^S and sends it to all the corrupted servers. For each corrupted server S, Sim_2 emulates each honest server to receive Ver^S and sends it to all the corrupted servers. Then, Sim_2 checks whether for each corrupted server S, the set Ver^S received by each honest server is the same. If not, Sim_2 aborts the protocol on behalf of each honest server. After completing the simulation, Sim_2 outputs what $\mathcal{A}$ outputs.

(b) For each server S and honest virtual server $V_j \in \mathsf{Ver}^S$ with $\mathsf{ROT}_j = 0$:

i. Sim_2 randomly samples $S_{j,1}$'s shares of all the sharings generated by an honest party in the sharing phase based on corrupted servers' shares except the additive sharings for V_j's shares of $[\boldsymbol{r}^{(i')}]_{\kappa'}, [\boldsymbol{r}^{(i')'}]^{(2)}_{\kappa'}, [\boldsymbol{o}^{(i')}]^{(2)}_{\kappa'}$ generated by the last honest server $S_{i'}, i' \in \mathcal{H}$. For $S_{j,1}$'s shares for the remaining 3 sharings, Sim_2 computes them based on his shares of $\sum_{j\in\mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i\in\mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle, \sum_{j\in\mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i\in\mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle, \sum_{j\in\mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V'_\alpha} \rangle + \sum_{i\in\mathcal{H}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle$ and $\sum_{j\in\mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i\in\mathcal{H}\setminus\{i'\}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle, \sum_{j\in\mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i\in\mathcal{H}\setminus\{i'\}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle, \sum_{j\in\mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V'_\alpha} \rangle + \sum_{i\in\mathcal{H}\setminus\{i'\}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle$.

ii. Sim_2 randomly samples a $(\kappa-1)$-bit string as $\Delta^{S_{j,1}}$.

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_2 randomly samples $\lambda_w^{S_{j,1}}, k_{w,0}^{S_{j,1}}$.

iv. For each input wire w of $\mathtt{Circ}^{V_j}$, Sim_2 computes $S_{j,1}$'s share $\lambda_w^{S_{j,1}}$ 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.(b).i.).

v. For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wires a, b and output wire o, Sim_2 computes

$$k_{o,0}^{S_{j,1}} \| \lambda_o^{S_{j,1}} = (k_{a,0}^{S_{j,1}} \| \lambda_a^{S_{j,1}}) \oplus (k_{b,0}^{S_{j,1}} \| \lambda_b^{S_{j,1}}).$$

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_2 computes $k_{w,1}^{S_{j,1}} = k_{w,0}^{S_{j,1}} \oplus \Delta^{S_{j,1}}$.

vii. For each execution of Π_{Mult} in the garbling phase, for each $i = 2, \ldots, c$, Sim_2 already has $r_1^{(1,i)} \oplus r_0^{(1,i)} \oplus x^{(1)}$, $(b^{(1,i)}, r_{b^{(1,i)}}^{(1,i)})$, $y^{(1)} \oplus r_{b^{(i,1)}}^{(i,1)}$, and $x^{(1)}, y^{(1)}$. With these values, Sim_2 computes $S_{j,1}$'s output from $\mathcal{F}_{\mathsf{ROT}}$.

viii. For each AND gate in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, Sim_2 computes $S_{j,1}$'s share of $\langle A_{g,2i_0+i_1}^{S_{j,\alpha}} \rangle$ for each $(i_0, i_1) = (0,0), (0,1), (1,0), (1,1)$ and $\alpha = 1, \ldots, c$ by following the computation process in the garbling phase.

ix. For each output gate in $\mathtt{Circ}^{V_j}$ with input wire w, Sim_2 computes $S_{j,1}$'s share of $\langle \mathsf{ct}_{w,i_2}^{S_{j,\alpha}} \rangle$ for each $i_2 = 0, 1$ and $\alpha = 1, \ldots, c$ by following the computation process in the garbling phase. Sim_2 knows all the elements in $\mathsf{IS}^{S_{j,1}}$ after completing the above steps.

x. Sim_2 sets $\mathsf{Check}_j = 1$.

(c) For each corrupted server S and honest virtual server $V_j \in \mathsf{Ver}^S$:

i. Then, Sim_2 emulates $\mathcal{F}_{\mathsf{Commit}}$ to send $(\mathsf{IS}^{S_{j,1}}, S_{j,1}, \tau_{\mathsf{IS}^{S_{j,1}}})$ to S and sends all the messages he emulates $S_{j,1}$ to send and receive in the garbling phase and Step 1 of the verification phase to S on behalf of $S_{j,1}$.

ii. For each honest party P, Sim_2 sends all the shares of sharings generated by P for $S_{j,1}, \ldots, S_{j,c}$ in the sharing phase to S on behalf of P.

(d) For each honest server S, Sim_2 emulates $\mathcal{F}_{\mathsf{Commit}}$ to receive $(\mathsf{open}, S_{j,i}, \tau_{\mathsf{IS}^{S_{j,i}}}, S)$ from each corrupted server $S_{j,i}$ for $V_j \in \mathsf{Ver}^S$. Then Sim_2 receives all the messages $S_{j,i}$ sends and receives in the garbling phase and Step 1 of the verification phase. For each corrupted party P, Sim_2 also receives all the shares generated by P for $S_{j,1}, \ldots, S_{j,c}$ in the sharing phase from P.

(e) For honest server S and each virtual server $V_j \in \mathsf{Ver}^S$, Sim_2 follows the protocol 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_2 aborts the protocol on behalf of S. After completing the simulation, Sim_2 outputs what $\mathcal{A}$ outputs.

(f) If $\mathsf{OTCheck} = 1$, Sim_2 aborts the simulation.

(g) If $\mathsf{CompCheck} = 1$, Sim_2 aborts the simulation.

(h) If $\mathsf{CorrCheck} = 1$, Sim_2 aborts the simulation.

(i) Sim_2 chooses a set $\mathcal{H}_{\mathsf{vir}}$ of $3N/4$ honest 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_2 checks whether the shares of the virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each Σ-sharing, $\Sigma^{(2)}$-sharing, and each $\Sigma^{(2)}$-sharing with an all-zero secret generated by a corrupted party except $[\boldsymbol{r}^{(i)}]_{\kappa'}, [\boldsymbol{r}^{(i)'}]_{\kappa'}^{(2)}, [\boldsymbol{o}^{(i)}]_{\kappa'}^{(2)}$ generated by corrupted server S_i is valid (i.e. there is a valid sharing that has the same shares for these virtual servers). If not, Sim_2 aborts the simulation. Otherwise, Sim_2 chooses a valid sharing as the sharing shared by the corrupted party and reconstructs the secret.

(j) For each honest virtual server V_j with $\mathsf{ROT}_j = \mathsf{Check}_j = 0$ out of $\mathcal{H}_{\mathsf{vir}}$:

i. Sim_2 randomly samples $S_{j,1}$'s shares of all the sharings generated by an honest party in the sharing phase based on the corrupted servers' shares except the additive sharings for V_j's shares of $[\boldsymbol{r}^{(i')}]_{\kappa'}, [\boldsymbol{r}^{(i')'}]_{\kappa'}^{(2)}, [\boldsymbol{o}^{(i')}]_{\kappa'}^{(2)}$ generated by the last honest server $S_{i'}, i' \in \mathcal{H}$. For $S_{j,1}$'s shares for the remaining 3 sharings, Sim_2 computes them based on his shares of $\sum_{j \in \mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle, \sum_{j \in \mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle, \sum_{j \in \mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle$ and $\sum_{j \in \mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H} \setminus \{i'\}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle, \sum_{j \in \mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H} \setminus \{i'\}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle, \sum_{j \in \mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H} \setminus \{i'\}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle$.

ii. Sim_2 randomly samples a $(\kappa - 1)$-bit string as $\Delta^{S_{j,1}}$.

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_2 randomly samples $\lambda_w^{S_{j,1}}, k_{w,0}^{S_{j,1}}$.

iv. For each input wire w of $\mathtt{Circ}^{V_j}$, Sim_2 computes $S_{j,1}$'s share $\lambda_w^{S_{j,1}}$ of $\langle\lambda\rangle$ 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.(j).i.).

v. For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, Sim_2 computes

$$k_{o,0}^{S_{j,1}} \| \lambda_o^{S_{j,1}} = (k_{a,0}^{S_{j,1}} \| \lambda_a^{S_{j,1}}) \oplus (k_{b,0}^{S_{j,1}} \| \lambda_b^{S_{j,1}}).$$

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_2 computes $k_{w,1}^{S_{j,1}} = k_{w,0}^{S_{j,1}} \oplus \Delta^{S_{j,1}}$.

vii. For each execution of Π_{Mult} in the garbling phase, for each $i = 2, \ldots, c$, Sim_2 already has $r_1^{(1,i)} \oplus r_0^{(1,i)} \oplus x^{(1)}$, $(b^{(1,i)}, r_{b^{(1,i)}}^{(1,i)})$, $y^{(1)} \oplus r_{b^{(i,1)}}^{(i,1)}$, and $x^{(1)}, y^{(1)}$. With these values, Sim_2 computes $S_{j,1}$'s output from $\mathcal{F}_{\mathsf{ROT}}$.

viii. For each AND gate in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, Sim_2 computes $S_{j,1}$'s share of $\langle A_{g,2i_0+i_1}^{S_{j,\alpha}}\rangle$ for each $(i_0, i_1) = (0,0), (0,1), (1,0), (1,1)$ and $\alpha = 1, \ldots, c$ by following the computation process in the garbling phase.

ix. For each output gate in $\mathtt{Circ}^{V_j}$ with input wire w, Sim_2 computes $S_{j,1}$'s share of $\langle \mathsf{ct}_{w,i_2}^{S_{j,\alpha}}\rangle$ for each $i_2 = 0, 1$ and $\alpha = 1, \ldots, c$ by following the computation process in the garbling phase.

3. Note that each output of $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ to a server receiver in Π_0 (which is a virtual server, say V_j) is shared by an additive sharing among $S_{j,1}, \ldots, S_{j,c}$, and all the secrets of these sharings have been computed by Sim_2 when $V_j \notin \mathcal{H}_{\mathsf{vir}}$. We can regard the secrets as the chosen output of corrupted servers from $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ (i.e. we regard that $V_j \notin \mathcal{H}_{\mathsf{vir}}$ are the corrupted servers of Π_0). In this way, Sim_2 constructs an adversary $\mathcal{A}'$ in Π_0 that interacts with honest parties in Π_0 with the secrets of these additive sharings while interacting with $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{input}}$ and then fail-stops before the evaluation phase. Sim_2 then invokes Sim_0 with adversary $\mathcal{A}'$. When Sim_0 invokes $\mathcal{F}$, Sim_2 sends the same message to $\mathcal{F}$. Then, Sim_2 gets the output of Sim_0.

Figure 37: The simulator for the verification phase of Π_2 when P_{king} is corrupted.

Simulator $\mathsf{Sim}_{2\text{-Eval}}$

Evaluation Phase

1. For each $i = 1, \ldots, \mathsf{rec}$, Sim_2 learns the shares for each $V_j \in \mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{s}_i]^{(2)}$ from the output of Sim_0. Sim_2 then runs the algorithm Alg_2 in Remark 1 to find a $\Sigma^{(2)}$-sharing as $[\boldsymbol{s}_i]^{(2)}$ for each $i = 1, \ldots, \mathsf{rec}$ such that the shares for ciortual servers in $\mathcal{H}_{\mathsf{vir}}$ matches the output from Sim_0. Then, Sim_2 does the following:
   (a) If the receiver R_i of the i-th reconstruction is an honest client, Sim_2 samples a random k-bit string as $\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}$ for each $\beta = 1, \ldots, c$ and $\alpha = 1, \ldots, \kappa$. Then, for $\alpha = 1, \ldots, \kappa$, Sim_2 randomly samples the whole sharings $[\boldsymbol{r}_{\boldsymbol{s}_i,1}^{(\alpha)}]^{(3)}, \ldots, [\boldsymbol{r}_{\boldsymbol{s}_i,c}^{(\alpha)}]^{(3)}$ based on the shares of the virtual servers not in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}]^{(3)} = [\boldsymbol{r}_{\mathbf{0},\beta}^{(\alpha)}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}_{\mathbf{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0},\beta}^{(\alpha)}]$ and the secret $\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}$ for each $\beta = 1, \ldots, c$.
   (b) If the receiver R_i of the i-th reconstruction is a virtual server in $\mathcal{H}_{\mathsf{vir}}$, Sim_2 samples a random κ-bit string as $\boldsymbol{r}_{s_{i,\eta},\eta,1} = (r_{s_{i,\eta},\eta,1,1}^{(1)}, \ldots, r_{s_{i,\eta},\eta,1}^{(\kappa)})$ 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 $\boldsymbol{r}_{\boldsymbol{s}_i,1}^{(\alpha)} = (r_{s_{i,1},1,1}^{(\alpha)}, r_{s_{i,2},2,1}^{(\alpha)}, \ldots, r_{s_{i,k},k,1}^{(\alpha)})$, Sim_2 randomly samples the whole sharings $[\boldsymbol{r}_{\boldsymbol{s}_i,1}^{(\alpha)}]^{(3)}$ based on the shares of the virtual servers not in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}_{\boldsymbol{s}_i,1}^{(\alpha)}]^{(3)} = [\boldsymbol{r}_{\mathbf{0},1}^{(\alpha)}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}_{\mathbf{1},1}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0},1}^{(\alpha)}]$ and the secret $\boldsymbol{r}_{\boldsymbol{s}_i,1}^{(\alpha)}$.
   (c) For each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$ and each $a = 1, \ldots, \ell^2$, let $s_{i,a}^{V_j}$ be the a-th bit of V_j's share of $[\boldsymbol{s}_i]$. Sim_2 computes $Y_{(i-1)\ell^2+a, s_{i,a}^{V_j}}^{S_{j,\beta}}$ for each $\beta = 1, \ldots, c$ with the shares of servers in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}_{\boldsymbol{s}_i,1}^{(\alpha)}]^{(3)}$ generated by the honest clients and the first servers emulating virtual servers in $\mathcal{H}_{\mathsf{vir}}$ and each pair of $[\boldsymbol{r}_{\mathbf{0},\beta}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\mathbf{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\mathbf{0},\beta}^{(\alpha)}]$ generated by other parties, where the computational process of each $Y_{(i-1)\ell^2+a, s_{i,a}^{V_j}}^{S_{j,\beta}}$ is the same as in the protocol with each $[\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}]^{(3)}$ generated by an honest client or a server $S_{j,1}$ for

$V_j \in \mathcal{H}_{\mathsf{vir}}$ being regarded as $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$.

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_2 samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string $k^{S_{j,1}}_{w,v_w\oplus\lambda_w}$. For each input wire of $\mathtt{Circ}^{V_j}$, Sim_2 samples a random $(\kappa-1)$-bit string $k^{S_{j,1}}_{w,v_w\oplus\lambda_w}$.

(b) For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wires a, b and output wire o, Sim_2 computes $k^{S_{j,1}}_{o,v_o\oplus\lambda_o} = k^{S_{j,1}}_{a,v_a\oplus\lambda_a} \oplus k^{S_{j,1}}_{b,v_b\oplus\lambda_b}$ 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_2 computes the ciphertext encrypted by $\{k^{S_{j,i}}_{a,v_a\oplus\lambda_a}, k^{S_{j,i}}_{b,v_b\oplus\lambda_b}\}^c_{i=1}$ by:

$$\Bigg(\bigoplus_{i=1}^{c} \big(\mathcal{O}_1(k^{S_{j,i}}_{a,v_a\oplus\lambda_a} \| (v_a \oplus \lambda_a) \| k^{S_{j,i}}_{b,v_b\oplus\lambda_b} \| (v_b \oplus \lambda_b) \| i \| j \| \beta \| g)\big)\Bigg) \oplus k^{S_{j,\beta}}_{o,v_o\oplus\lambda_o} \| (v_o \oplus \lambda_o).$$

Then Sim_2 samples 3 random κ-bit strings as the other 3 ciphertexts for this gate g and each $\beta = 1, \ldots, c$.

(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 can be obtained from the output of Sim_0. Sim_2 sets the output wire value v_w to be the corresponding bit from the output of Sim_0. Then, Sim_2 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_2 computes

$$\mathsf{ct}^{S_{j,\beta}}_{w,v_w\oplus\lambda_w} = \Bigg(\bigoplus_{i=1}^{c} \mathcal{O}_2(k^{S_{j,i}}_{w,v_w\oplus\lambda_w} \| (v_w \oplus \lambda_w) \| i \| j \| \beta \| w)\Bigg) \oplus Y^{S_{j,\beta}}_{\mathsf{k},\lambda_w}.$$

Then Sim_2 samples a random $c\ell\kappa$-bit string as the other ciphertext for this wire w and each $\beta = 1, \ldots, c$.

2. **Sending Output Masks.** For each input wire w of an output gate in $\mathtt{Circ}^{V_j}$ for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 computes $S_{j,1}$'s share of $\langle\lambda_w\rangle$ based on the secret and $S_{j,2}, \ldots, S_{j,c}$'s shares. Then Sim_2 sends it to P_{king} on behalf of $S_{j,1}$.

3. **Encrypting Input Labels.** For each $i = 1, \ldots, \mathsf{rec}$, if R_i is a virtual server V_j in $\mathcal{H}_{\mathsf{vir}}$ and the η-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_η in R_i's circuit $\mathtt{Circ}^{V_j}$, Sim_2 computes

$$\mathsf{ct}^{(i,1)}_{j_\eta,s_{i,\eta}} = \mathcal{O}_1(\boldsymbol{r}_{s_{i,\eta},\eta,1} \| s_{i,\eta} \| i \| 1 \| \eta \| j_\eta) \oplus \big(k^{S_{j,1}}_{w_{j_\eta},v_{w_{j_\eta}}\oplus\lambda_{w_{j_\eta}}} \| (v_{w_{j_\eta}} \oplus \lambda_{w_{j_\eta}})\big).$$

Then, Sim_2 samples a random κ-bit string as $\mathsf{ct}^{(i,1)}_{j_\eta,1\oplus s_{i,\eta}}$ and sends $\{\mathsf{ct}^{(i,1)}_{j_\eta,0}, \mathsf{ct}^{(i,1)}_{j_\eta,1}\}$ to P_{king} on behalf of $S_{j,1}$.

4. **Sending Input Labels.** For each input wire w of each honest virtual server V_j's local circuit $\mathtt{Circ}^{V_j}$, if the wire value x_w doesn't come from a reconstruction of a $\Sigma^{(2)}$-sharing Sim_2 sends $x_w \oplus \lambda_w$ (which has been computed in the garbling phase) and $k^{S_{j,1}}_{w,x_w\oplus\lambda_w}$ to P_{king} on behalf of $S_{j,1}$.

5. **Sending Garbled Circuits.** For each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 computes $S_{j,1}$'s shares of $\langle\boldsymbol{A}_g\rangle, \langle\boldsymbol{B}_g\rangle, \langle\boldsymbol{C}_g\rangle, \langle\boldsymbol{D}_g\rangle$ for each AND gate in $\mathtt{Circ}^{V_j}$ and $\langle\mathbf{ct}_{w,0}\rangle, \langle\mathbf{ct}_{w,1}\rangle$ for each output wire w of $\mathtt{Circ}^{V_j}$ based on the secrets and corrupted servers' shares. Then for each honest virtual server V_j, Sim_2 sends these shares to P_{king} on behalf of $S_{j,1}$.

6. **Sending Outputs.** For each receiver R_i that is an honest client:

(a) Sim_2 receives $\boldsymbol{s}_i$ and $\{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}\}^\kappa_{\alpha=1}$ for each $\beta = 1, \ldots, c$ from P_{king}.

(b) Sim_2 checks whether $\boldsymbol{s}_i$ and $\{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}\}^\kappa_{\alpha=1}$ for each $\beta = 1, \ldots, c$ from P_{king} are all correctly sent. If not, Sim_2 aborts the protocol on behalf of R_i.

7. Sim_2 outputs what $\mathcal{A}$ outputs.

Figure 38: The simulator for the evaluation phase of Π_2 when P_{king} is corrupted.

We construct the following hybrids:

$\mathbf{Hyb}_0$: In this hybrid, Sim_2 gets honest clients' 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_2 aborts the simulation if over $N/16$ virtual servers are corrupted. By Chernoff bound, if the virtual servers are truly randomly determined, the probability that over $N/16$ virtual servers are corrupted is negligible. From the definition of a PRG, the probability that pseudorandom sets of servers emulating the virtual servers also satisfy that over $N/16$ virtual servers are corrupted is also negligible, or the result of the PRG can be distinguished from truly random party sets with a non-negligible probability by checking whether over $N/16$ virtual servers are corrupted. Thus, the distributions of $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ are computationally indistinguishable.

$\mathbf{Hyb}_2$: 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_2 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 servers who act as a virtual server, Sim_2 first generates the corrupted servers' shares, and then randomly samples the honest servers' shares based on corrupted servers' shares and the secret. Since for all these sharings, the set of corrupted servers' shares is independent of the secret, we only change the order of generating the honest and corrupted servers' 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_2 additionally sets $\mathsf{CompCheck} = \mathsf{CorrCheck} = \mathsf{OTCheck} = 0$ at the beginning of the simulation. Then, for each honest virtual server V_j, Sim_2 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, after the corrupted servers commit their local inputs, Sim_2 additionally performs the following checks:

1. For each sharing generated by an honest party, for each honest virtual server V_j, let the generated share of V_j be s. Sim_2 checks whether the committed inputs of $S_{j,2}, \ldots, S_{j,c}$'s shares of $\langle s \rangle$ matches what Sim_2 sends to them. If the check of V_j fails, Sim_2 sets $\mathsf{Corr}_j = 1$.

2. For the result of $\mathcal{F}_{\mathsf{ROT}}$, Sim_2 checks whether for each honest server V_j, the corrupted server $S_{j,2}, \ldots, S_{j,c}$ all committed their outputs correctly. For each honest virtual server V_j that fails in the check, Sim_2 sets $\mathsf{ROT}_j = 1$.

3. If for at least $N/12$ honest virtual server V_j it holds that $\mathsf{ROT}_j = 1$, Sim_2 sets $\mathsf{OTCheck} = 1$ and takes the first $N/12$ of them. For each of the other honest virtual servers V_j with $\mathsf{ROT}_j = 1$, Sim_2 sets $\mathsf{ROT}_j = 0$.

4. If there are over $N/48$ honest virtual servers V_j with $\mathsf{Corr}_j = 1$, Sim_2 sets $\mathsf{CorrCheck} = 1$.

This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same output distribution.

$\mathbf{Hyb}_6$: In this hybrid, for each sharing generated by a corrupted party in the sharing phase, Sim_2 regarded that all the corrupted servers correctly commit the shares they get, and then Sim_2 can reconstruct the shares of honest virtual servers for each $\Sigma, \Sigma^{(2)}$-sharing generated by a corrupted party. Since the messages between corrupted parties can be regarded as anything, we don't change 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 with $\mathsf{ROT}_j = 0$, for each execution of Π_{Mult}, Sim_2 doesn't follow the protocol to compute $r_1^{(1,i)} \oplus r_0^{(1,i)} \oplus x^{(1)}$ and $y^{(1)} \oplus b^{(i,1)}$ for each $i = 2, \ldots, c$. Instead, Sim_2 randomly samples a $(\kappa - 1)$-bit string as $r_1^{(1,i)} \oplus r_0^{(1,i)} \oplus x^{(1)}$ and a random bit as $y^{(1)} \oplus b^{(i,1)}$. Then, Sim_2 computes $r_1^{(1,i)} \oplus r_0^{(1,i)}$ based on $r_1^{(1,i)} \oplus r_0^{(1,i)} \oplus x^{(1)}$ and $x^{(1)}$ and then

determines $(r_0^{(1,i)}, r_1^{(1,i)})$ based on $(b^{(1,i)}, r_{b^{(1,i)}}^{(1,i)})$. Similarly, Sim_2 computes $b^{(i,1)}$ based on $y^{(1)} \oplus b^{(i,1)}$ and $y^{(1)}$ to determine $r_{b^{(i,1)}}^{(i,1)}$. Since $(r_0^{(1,i)}, r_1^{(1,i)})$ are sampled randomly based on $(b^{(1,i)}, r_{b^{(1,i)}}^{(1,i)})$, $r_1^{(1,i)} \oplus r_0^{(1,i)} \oplus x^{(1)}$ is a random $(\kappa-1)$-bit string. Since $b^{(i,1)}$ is a random bit, so is $y^{(1)} \oplus b^{(i,1)}$. Thus, we only change the order of generating $r_1^{(1,i)} \oplus r_0^{(1,i)} \oplus x^{(1)}$ and $(r_0^{(1,i)}, r_1^{(1,i)})$, $y^{(1)} \oplus b^{(i,1)}$ and $b^{(i,1)}$ 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 servers emulating an honest virtual server V_j with $\mathsf{ROT}_j = 0$. Sim_2 additionally checks whether $r_1^{(i,1)} \oplus r_0^{(i,1)} \oplus x^{(i)}$ and $y^{(i)} \oplus b^{(1,i)}$ from $S_{j,i}$ for each $i = 2, \ldots, c$ are correctly computed with their committed inputs. If not, Sim_2 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 with $\mathsf{ROT}_j = 0$ and each input wire w of $\mathtt{Circ}^{V_j}$, Sim_2 doesn't follow the protocol to compute $S_{j,1}$'s share of $\langle x_w \oplus \lambda_w \rangle$. Instead, Sim_2 samples a random bit as $S_{j,1}$'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 $S_{j,2}, \ldots, S_{j,c}$'s shares of $\langle x_w \oplus \lambda_w \rangle$, Sim_2 additionally checks whether they match the inputs they committed. If not, Sim_2 sets $\mathsf{Comp}_j = 1$. Since $S_{j,1}$'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, Sim_2 doesn't follow the protocol to compute $S_{\alpha,1}$'s shares of $\langle \sum_{j=1}^{k_1} s_j \cdot \boldsymbol{x}_j^{V_\alpha} + \sum_{i=1}^{n} \boldsymbol{r}^{(i,V_\alpha)} \rangle$, $\langle \sum_{j=1}^{k_2} s'_j \cdot \boldsymbol{x}_j^{V'_\alpha} + \sum_{i=1}^{n} \boldsymbol{r}^{(i,V_\alpha)'} \rangle$, and $\langle \sum_{j=1}^{k_3} s^{(j)} \cdot \boldsymbol{o}_j^{V_\alpha} + \sum_{i=1}^{n} \boldsymbol{o}^{(i,V_\alpha)} \rangle$ for each honest virtual server V_α. For the last honest server $S_{i'} \in \mathcal{H}$, Sim_2 doesn't compute $S_{\alpha,1}$'s shares of $\langle \boldsymbol{r}^{(i',V_\alpha)} \rangle$, $\langle \boldsymbol{r}^{(i',V_\alpha)'} \rangle$, and $\langle \boldsymbol{o}^{(i',V_\alpha)} \rangle$. Sim_2 computes $\sum_{j \in \mathcal{C}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle, \sum_{j \in \mathcal{C}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle, \sum_{j \in \mathcal{C}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle$ and $\sum_{j \in \mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H} \setminus \{i'\}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle, \sum_{j \in \mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H} \setminus \{i'\}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle, \sum_{j \in \mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H} \setminus \{i'\}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle$ first (corrupted servers' shares are computed with the shares they receive in the sharing phase). Then, for $\sum_{j \in \mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle, \sum_{j \in \mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle, \sum_{j \in \mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle$, Sim_2 randomly samples the secrets for each honest virtual server V_α 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 $\Sigma_{\kappa'}^{(2)}$-sharing with an all-zero secret respectively. Then, Sim_2 computes $S_{\alpha,1}$'s shares of $\sum_{j \in \mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle, \sum_{j \in \mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle, \sum_{j \in \mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle$ based on the secret and $S_{\alpha,2}, \ldots, S_{\alpha,c}$'s shares (which are computed with their shares generated by Sim_2 in the sharing phase). For the last honest server $S_{i'}, i' \in \mathcal{H}$, Sim_2 computes each $S_{\alpha,1}$'s shares of $\langle \boldsymbol{r}^{(i',V_\alpha)} \rangle$, $\langle \boldsymbol{r}^{(i',V_\alpha)'} \rangle$, and $\langle \boldsymbol{o}^{(i',V_\alpha)} \rangle$ based on his shares of $\sum_{j \in \mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle, \sum_{j \in \mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle, \sum_{j \in \mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle$ and $\sum_{j \in \mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H} \setminus \{i'\}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle, \sum_{j \in \mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H} \setminus \{i'\}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle, \sum_{j \in \mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H} \setminus \{i'\}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle$. Finally, Sim_2 computes $S_{\alpha,1}$'s share of $\langle \sum_{j=1}^{k_1} s_j \cdot \boldsymbol{x}_j^{V_\alpha} + \sum_{i=1}^{n} \boldsymbol{r}^{(i,V_\alpha)} \rangle$, $\langle \sum_{j=1}^{k_2} s'_j \cdot \boldsymbol{x}_j^{V'_\alpha} + \sum_{i=1}^{n} \boldsymbol{r}^{(i,V_\alpha)'} \rangle$, and $\langle \sum_{j=1}^{k_3} s^{(j)} \cdot \boldsymbol{o}_j^{V_\alpha} + \sum_{i=1}^{n} \boldsymbol{o}^{(i,V_\alpha)} \rangle$ by

$$\sum_{j=1}^{k_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i=1}^{n} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle = \sum_{j \in \mathcal{C}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle + \sum_{j \in \mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)} \rangle,$$

$$\sum_{j=1}^{k_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i=1}^{n} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle = \sum_{j \in \mathcal{C}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle + \sum_{j \in \mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{r}^{(i,V_\alpha)'} \rangle,$$

and

$$\sum_{j=1}^{k_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i=1}^{n} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle = \sum_{j \in \mathcal{C}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{C}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle + \sum_{j \in \mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha} \rangle + \sum_{i \in \mathcal{H}} \langle \boldsymbol{o}^{(i,V_\alpha)} \rangle.$$

Since the sharings $[\boldsymbol{r}^{(i')}]_{\kappa'}, [\boldsymbol{r}^{(i')'}]^{(2)}_{\kappa'}, [\boldsymbol{o}^{(i')}]^{(2)}_{\kappa'}$ are randomly generated based on the corrupted virtual servers' shares, we can regard that Sim_2 emulates $S_{i'}$ to generate the secrets for each corrupted virtual server V_α first and then generate the shares for each honest virtual server V_α. Thus, the secrets of $\sum_{j\in\mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha}\rangle + \sum_{i\in\mathcal{H}}\langle \boldsymbol{r}^{(i,V_\alpha)}\rangle$ (resp. $\langle \sum_{j=1}^{k_2} s'_j \cdot \boldsymbol{x}_j^{V'_\alpha} + \sum_{i=1}^{n} \boldsymbol{r}^{(i,V_\alpha)'}\rangle$ and $\langle \sum_{j=1}^{k_3} s^{(j)} \cdot \boldsymbol{o}_j^{V_\alpha} + \sum_{i=1}^{n} \boldsymbol{o}^{(i,V_\alpha)}\rangle$), which is computed by adding honest virtual server V_α's share of $[\boldsymbol{r}^{(i')}]_{\kappa'}$ (resp. $[\boldsymbol{r}^{(i')'}]^{(2)}_{\kappa'}$ and $[\boldsymbol{o}^{(i')}]^{(2)}_{\kappa'}$) is also random when those secrets for corrupted virtual servers are fixed. Thus, we only change the order of generating $S_{\alpha,1}$'s shares of $\langle \boldsymbol{r}^{(i',V_\alpha)}\rangle, \langle \boldsymbol{r}^{(i',V_\alpha)'}\rangle, \langle \boldsymbol{o}^{(i',V_\alpha)}\rangle$ and $\sum_{j\in\mathcal{H}_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha}\rangle + \sum_{i\in\mathcal{H}}\langle \boldsymbol{r}^{(i,V_\alpha)}\rangle, \sum_{j\in\mathcal{H}_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha}\rangle + \sum_{i\in\mathcal{H}}\langle \boldsymbol{r}^{(i,V_\alpha)'}\rangle, \sum_{j\in\mathcal{H}_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha}\rangle + \sum_{i\in\mathcal{H}}\langle \boldsymbol{o}^{(i,V_\alpha)}\rangle$ for each honest virtual server V_α without changing their distributions. Thus, $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ have the same output distribution.

$\mathbf{Hyb}_{11}$: In this hybrid, during the verification phase, Sim_2 additionally sets $\mathsf{Comp}_\alpha = 1$ if V_α is an honest virtual server with $\mathsf{ROT}_\alpha = 0$ but the corrupted servers $S_{\alpha,2}, \ldots, S_{\alpha,c}$'s shares of $\sum_{j=1}^{k_1} s_j \cdot \langle \boldsymbol{x}_j^{V_\alpha}\rangle + \sum_{i=1}^{n}\langle \boldsymbol{r}^{(i,V_\alpha)}\rangle$, $\sum_{j=1}^{k_2} s'_j \cdot \langle \boldsymbol{x}_j^{V'_\alpha}\rangle + \sum_{i=1}^{n}\langle \boldsymbol{r}^{(i,V_\alpha)'}\rangle$, or $\sum_{j=1}^{k_3} s^{(j)} \cdot \langle \boldsymbol{o}_j^{V_\alpha}\rangle + \sum_{i=1}^{n}\langle \boldsymbol{o}^{(i,V_\alpha)}\rangle$ received by some honest server don't match the values they commit. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{11}$ and $\mathbf{Hyb}_{10}$ have the same output distribution.

$\mathbf{Hyb}_{12}$: In this hybrid, during the verification phase, at the end of the verification of the sharings, Sim_2 sets $\mathsf{CompCheck} = 1$ if there are over $N/48$ honest virtual servers V_j with $\mathsf{Comp}_j = 1$. This doesn't affect the output distribution. In addition, If for two different honest servers, the sharings $[\tau]_{\kappa'}, [\tau']^{(2)}_{\kappa'}, [\tau_0]^{(2)}_{\kappa'}$ computed on behalf of them are different but $\mathcal{O}_1([\tau]_{\kappa'}, [\tau']^{(2)}_{\kappa'}, [\tau_0]^{(2)}_{\kappa'})$ are the same, Sim_2 aborts the simulation. Since Sim_2 emulates the random oracle honestly, this only happens when there exists two queries q_1, q_2 among $\mathsf{poly}(\kappa)$ queries to $\mathcal{O}_1$ (either by the adversary or by honest parties) such that $\mathcal{O}_1(q_1) = \mathcal{O}_1(q_2)$. This only happens with a negligible probability. Thus, $\mathbf{Hyb}_{12}$ and $\mathbf{Hyb}_{11}$ are computationally indistinguishable.

$\mathbf{Hyb}_{13}$: In this hybrid, during the verification phase, for each honest server, Sim_2 doesn't follow the protocol to check each honest virtual server V_j's local computation. Instead, Sim_2 checks whether $S_{j,2}, \ldots, S_{j,c}$ perform their computation and send the messages in the garbling phase and Step 1 of the verification phase honestly with their committed inputs. Sim_2 also checks whether V_j's shares of $[\boldsymbol{r}^{(i)'}]_{\kappa'}, [\boldsymbol{r}^{(i)'}]^{(2)}_{\kappa'}, [\boldsymbol{o}^{(i)}]^{(2)}_{\kappa'}$ for each corrupted server S_i are correctly sent. In addition, Sim_2 checks whether the committed outputs from $\mathcal{F}_{\mathsf{ROT}}$ are valid. If any of the checks fail, Sim_2 aborts the protocol on behalf of S. Note that the only difference between the two hybrids is that Sim_2 doesn't check the computation of honest servers. Since honest servers always follow the protocol to perform computation, this doesn't change the output distribution. Thus, $\mathbf{Hyb}_{13}$ and $\mathbf{Hyb}_{12}$ have the same output distribution.

$\mathbf{Hyb}_{14}$: In this hybrid, during the verification phase, after doing the checks, if $\mathsf{OTCheck} = 1$, Sim_2 aborts the simulation. This only changes the distribution when some corrupted server among $S_{j,2}, \ldots, S_{j,c}$ doesn't correctly commit his output from $\mathcal{F}_{\mathsf{ROT}}$ for over $N/12$ honest virtual parties V_j after simulating the sharing phase but the verification passes. Then, there must be $N/12$ honest virtual servers V_j with $\mathsf{ROT}_j = 1$. The probable ways that there is a $V_j \in \mathsf{Ver}^S$ of $\mathsf{ROT}_j = 1$ that passes the check are

1. For the execution of $\mathcal{F}_{\mathsf{ROT}}$ between $(S_{j,1}, S_{j,i})$, $S_{j,i}$ sends $b^{(1,i)}, r^{(1,i)}_{b^{(1,i)}}$ to $\mathcal{F}_{\mathsf{ROT}}$ but he correctly commits $(1 \oplus b^{(1,i)}, r^{(1,i)}_{1\oplus b^{(1,i)}})$ as his output from $\mathcal{F}_{\mathsf{ROT}}$.

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

For the first way, the adversary should commit $r^{(1,i)}_{1\oplus b^{(1,i)}}$, which is a random $(\kappa-1)$-bit string that is not used in any computation of messages before $S_{j,i}$ commits it, and Sim_2 can sample $r^{(1,i)}_{1\oplus b^{(1,i)}}$ after the adversary does the commitment. Thus, the adversary correctly commits $r^{(1,i)}_{1\oplus b^{(1,i)}}$ with a negligible probability $2^{\kappa-1}$. For the second way, the adversary should guess the random bit $b^{(1,i)}$ correctly. Since this bit is not used in any computation of messages before $S_{j,i}$ sends the commitments, Sim_2 can sample it after the adversary does the commitment. Thus, the adversary correctly commits $r^{(1,i)}_{b^{(1,i)}}$ and another value which is not equal to

$r^{(1,i)}_{1\oplus b^{(1,i)}}$ with a probability 1/2. Then, for each honest virtual server $V_j \in \mathsf{Ver}^S$ of $\mathsf{ROT}_j = 1$, the probability that the behavior of the adversary is among the above two ways is no more than 1/2.

Recall that there are $N/12$ honest virtual servers V_j with $\mathsf{ROT}_j = 1$, the expectation of the number of virtual servers V_j with $\mathsf{ROT}_j = 1$ that can pass the check is no more than $N/24$. By Chernoff bound, the probability that fewer than $N/48$ virtual servers V_j with $\mathsf{ROT}_j = 1$ fail to pass the check is no more than $e^{-N/192}$. When there are at least $N/48$ virtual servers V_j with $\mathsf{ROT}_j = 1$ that fail to pass the check, the probability that each honest server doesn't choose these virtual servers is

$$\frac{\frac{47N}{48}}{N} \cdot \frac{\frac{47N}{48}-1}{N-1} \cdot \cdots \cdot \frac{\frac{47N}{48}-\frac{N}{16n}+1}{N-\frac{N}{16n}+1} > \left(1-\frac{1}{48}\right)^{\frac{N}{16n}}.$$

Thus, the probability that these virtual servers are not chosen by all the honest servers is $(47/48)^{(\epsilon N/16)}$. Recall that $N = \Theta(n+\kappa)$, the probability is also negligible.

Thus, the distributions of $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{13}$ are statistically close.

$\mathbf{Hyb}_{15}$: In this hybrid, during the verification phase, after doing the checks, if $\mathsf{CompCheck} = 1$, Sim_2 aborts the simulation. This only changes the distribution when $\mathsf{Comp}_j = 1$ for over $N/16$ indices $j \in \{1,\ldots,N\}$ but the verification passes with $\mathsf{CompCheck} = 0$. Note that $\mathsf{Comp}_j = 1$ only happens when V_j is an honest virtual server and some message sent by $S_{j,2},\ldots,S_{j,c}$ doesn't match their committed inputs. Thus, if $V_j \in \mathsf{Ver}^S$ for some honest server S, the check won't pass, so the distribution only changes when there are over $N/48$ indices $j \in \{1,\ldots,N\}$ with $\mathsf{Comp}_j = 1$ but each V_j of them is not in any $V_j \in \mathsf{Ver}^S$ for some honest server S. The probability that each honest server doesn't choose these virtual servers is

$$\frac{\frac{47N}{48}}{N} \cdot \frac{\frac{47N}{48}-1}{N-1} \cdot \cdots \cdot \frac{\frac{47N}{48}-\frac{N}{16n}+1}{N-\frac{N}{16n}+1} > \left(1-\frac{1}{48}\right)^{\frac{N}{16n}}.$$

Thus, the probability that the $N/48$ virtual servers are not chosen by all the honest servers is $(47/48)^{(\epsilon N/16)}$. Recall that $N = \Theta(n+\kappa)$, the probability is negligible. Thus, the distributions of $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$ are statistically close.

$\mathbf{Hyb}_{16}$: In this hybrid, during the verification phase, after doing the checks, if $\mathsf{CorrCheck} = 1$, Sim_2 aborts the simulation. This only changes the distribution when $\mathsf{Corr}_j = 1$ for over $N/48$ indices $j \in \mathcal{H}$ after simulating the sharing phase 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 $S_{j,2},\ldots,S_{j,c}$ of $\langle s^{V_\alpha}\rangle$ doesn't match the shares of $\langle s^{V_\alpha}\rangle$ sent to them for V_j's share s^{V_α} of a Σ or $\Sigma^{(2)}$ generated by an honest party.

If over $N/32$ honest virtual servers V_j have $\mathsf{Corr}_j = 1$ because of this reason, then if $V_j \in \mathsf{Ver}^S$ for some honest server S, the check won't pass. Thus, the distribution only changes when there are over $N/48$ indices $j \in \{1,\ldots,N\}$ with $\mathsf{Corr}_j = 1$ but each V_j of them is not in any $V_j \in \mathsf{Ver}^S$ for some honest server S. The probability that each honest server doesn't choose these virtual servers is

$$\frac{\frac{47N}{48}}{N} \cdot \frac{\frac{47N}{48}-1}{N-1} \cdot \cdots \cdot \frac{\frac{47N}{48}-\frac{N}{16n}+1}{N-\frac{N}{16n}+1} > \left(1-\frac{1}{48}\right)^{\frac{N}{16n}}.$$

Thus, the probability that the $N/48$ virtual servers that are not chosen by all the honest servers is $(47/48)^{(\epsilon N/16)}$. 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, Sim_2 chooses a set $\mathcal{H}_{\mathsf{vir}}$ of $3N/4$ 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_2 checks whether the shares of the virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each Σ-sharing, $\Sigma^{(2)}$-sharing, and each $\Sigma^{(2)}$-sharing with an all-zero secret generated by a corrupted party except $[\boldsymbol{r}^{(i)}]_{\kappa'}, [\boldsymbol{r}^{(i)\prime}]^{(2)}_{\kappa'}, [\boldsymbol{o}^{(i)}]^{(2)}_{\kappa'}$ generated by corrupted server S_i is valid. If not, Sim_2 aborts the simulation. Otherwise, Sim_2 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 $[\tau]_{\kappa'} = \sum_{j=1}^{k_1} s_j \cdot [\boldsymbol{x}_j]_{\kappa'} + \sum_{i=1}^{n} [\boldsymbol{r}^{(i)}]_{\kappa'}$ 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 $\binom{N}{3N/4} < 2^N$ 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. Thus, if there is a non-negligible probability that $[\tau]_{\kappa'}$ is valid, then the truly random field elements and the pseudo-random values $s_1, \ldots, s_{k_1} \in \mathbb{F}_{2^{\kappa'}}$ can be distinguished by computing $[\tau]_{\kappa'}$ with a non-negligible probability, which contradicts the definition of a PRG, so the probability that $[\tau]_{\kappa'}$ is valid is negligible.

Similarly, when the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of a $\Sigma^{(2)}$-sharing or a $\Sigma^{(2)}$-sharing with an all-zero secret is not valid, the verification only passes with a negligible probability. Therefore, the distribution only changes with a negligible probability. Thus, the distributions of $\mathbf{Hyb}_{17}$ and $\mathbf{Hyb}_{16}$ are computationally indistinguishable.

$\mathbf{Hyb}_{18}$: 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_2 does not generate them at the beginning of the sharing phase. Instead, for those honest virtual servers V_j with $\mathsf{ROT}_j = 1$, Sim_2 generates the shares for them at the end of the sharing phase. For other honest virtual servers V_j that are in Ver^S for some server S, V_j's shares of these sharings are generated based on the corrupted servers' shares after Ver^S is verified. For other honest virtual servers out of $\mathcal{H}_{\mathsf{vir}}$, Sim_2 generates V_j's shares of these sharings based on the checked honest virtual servers' shares and the corrupted virtual servers' shares. For the honest virtual servers in $\mathcal{H}_{\mathsf{vir}}$, Sim_2 generates their shares based on other virtual servers' shares after the verification phase (at the beginning of the evaluation phase). Finally, Sim_2 computes the shares of $S_{j,1}$'s shares of the additive sharings of these shares based on $S_{j,2}, \ldots, S_{j,c}$'s shares and the secrets. Besides, Sim_2 samples the local randomness of $S_{j,1}$ in emulating each $V_j \in \mathcal{H}_{\mathsf{vir}}$ after the verification phase instead of in the sharing phase, and the computation of $S_{j,1}$ 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 out of $\mathcal{H}_{\mathsf{vir}}$ are independent of the secret, first sampling the shares of honest virtual servers out of $\mathcal{H}_{\mathsf{vir}}$ and then sampling the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ based on corrupted servers' shares and the secret won't affect the output distribution. Besides, the shares and randomness for $S_{j,1}$ emulating each virtual server V_j in $\mathcal{H}_{\mathsf{vir}}$ are not used in the simulation before the evaluation phase. Therefore, delaying the generation won't affect the output distribution. Thus, $\mathbf{Hyb}_{18}$ and $\mathbf{Hyb}_{17}$ have the same output distribution.

$\mathbf{Hyb}_{19}$: 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_2 additionally follows the execution of Π_0 (where each reconstruction of $\Sigma^{(2)}$-sharing is done from the shares of virtual servers in $\mathcal{H}_{\mathsf{Vir}}$) to compute the value v_w of w. Then, Sim_2 runs the algorithm Alg_2 in Remark 1 with the shares for virtual servers in $\mathcal{H}_{\mathsf{Vir}}$ of $[\boldsymbol{s}_i]^{(2)}$ for each $i = 1, \ldots, \mathsf{rec}$ with an honest receiver (either an honest client or a virtual server in $\mathcal{H}_{\mathsf{Vir}}$) to decide the whole sharing $[\boldsymbol{s}_i]^{(2)}$. If the receiver is an honest client, Sim_2 uses them to compute shares for virtual servers out of $\mathcal{H}_{\mathsf{Vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ for each $\alpha = 1, \ldots, \kappa$ and $\beta = 1, 2, 3$. If the receiver is a virtual server in $\mathcal{H}_{\mathsf{Vir}}$, Sim_2 uses them to compute shares for virtual servers out of $\mathcal{H}_{\mathsf{Vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},1}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},1} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},1}]$ for each $\alpha = 1, \ldots, \kappa$. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{19}$ and $\mathbf{Hyb}_{18}$ have the same output distribution.

$\mathbf{Hyb}_{20}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$:

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

   - If R_i is an honest client, Sim_2 doesn't generate the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ 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_2 follows the protocol to generate each $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$ first and then samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ based on the shares for virtual servers out of $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$. Then,

Sim_2 computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}, [\boldsymbol{s}_i]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ for each $\beta = 1, \ldots, c$.

- If R_i is a virtual server in $\mathcal{H}_{\mathsf{vir}}$, Sim_2 doesn't generate the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},1}]^{(3)}$ first and then computes their shares of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}]^{(3)}$ by himself. Instead, Sim_2 follows the protocol to generate each $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}$ first and then samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}]^{(3)}$ based on the shares for virtual servers out of $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}]^{(3)}$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}$. Then, Sim_2 computes the shares for virtual servers out of $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},1}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,1}]^{(3)}, [\boldsymbol{s}_i]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},1} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},1}]$.
- If R_i is a virtual server out of $\mathcal{H}_{\mathsf{vir}}$, Sim_2 just follows the protocol to receive honest servers' shares from corrupted parties.

Then Sim_2 follows the computation process of each $Y^{S_{j,\beta}}_{(i-1)\ell^2+a, s^{V_j}_{i,a}}$ in the protocol to compute each $Y^{S_{j,\beta}}_{(i-1)\ell^2+a, s^{V_j}_{i,a}}$ 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 $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$. For the other label $Y^{S_{j,\beta}}_{(i-1)\ell^2+a, 1\oplus s^{V_j}_{i,a}}$, Sim_2 still follows the protocol to compute it.

2. For 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_2 samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string as $k^{S_{j,1}}_{w, v_w \oplus \lambda_w}$. For each input wire of $\mathtt{Circ}^{V_j}$, Sim_2 samples a random $(\kappa-1)$-bit string as $k^{S_{j,1}}_{w, v_w \oplus \lambda_w}$. Then Sim_2 computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$ for all these wires. After all the ciphertexts are generated, Sim_2 samples a random $(\kappa-1)$-bit string as $\Delta^{S_{j,1}}$. Then, for each wire w in $\mathtt{Circ}^{V_j}$ that is not an output wire of an output gate, Sim_2 computes $k^{S_{j,1}}_{w, 1\oplus v_w \oplus \lambda_w} = k^{S_{j,1}}_{w, v_w \oplus \lambda_w} \oplus \Delta^{S_{j,1}}$.

3. Sim_2 maintains a set Q_1. For each AND gate g in $\mathtt{Circ}^{V_j}$ with input wire a, b, when the server $S_{j,1}$ computes his shares of the additive sharings of the ciphertexts of each gate except those computed with $\{k^{S_{j,i}}_{a, v_a \oplus \lambda_a}, k^{S_{j,i}}_{b, v_b \oplus \lambda_b}\}^c_{i=1}$, Sim_2 checks whether the query to the random oracle $\mathcal{O}_1$ has been queried before. If true, Sim_2 aborts the simulation. Otherwise, Sim_2 adds the query to Q_1.

4. Sim_2 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 server $S_{j,1}$ computes his shares of the additive sharings of $\mathsf{ct}^{S_{j,\beta}}_{w,i_2}$ for each $\beta = 1, \ldots, c$, Sim_2 checks whether the query to the random oracle $\mathcal{O}_2$ has been queried before. If true, Sim_2 aborts the simulation. Otherwise, Sim_2 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_2 doesn't follow the protocol to compute $S_{j,1}$'s shares of the additive sharings of the ciphertexts except $A^{S_{j,\beta}}_{g, 2(v_a \oplus \lambda_a) + (v_b \oplus \lambda_b)}$ for $\beta = 1, \ldots, c$. Instead, Sim_2 samples 3 random κ-bit strings as the ciphertexts and computes $S_{j,1}$'s shares of the additive sharings of all the ciphertexts based on the secrets and $S_{j,2}, \ldots, S_{j,c}$'s shares. Sim_2 computes the output of $\mathcal{O}_1$ 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_2 doesn't follow the protocol to compute $S_{j,1}$'s shares of the additive sharings of the ciphertexts $\mathsf{ct}^{S_{j,\beta}}_{w, 1\oplus v_w \oplus \lambda_w}$ for $\beta = 1, \ldots, c$. Instead, Sim_2 samples a random $c\ell\kappa$-bit string as the ciphertexts and computes $S_{j,1}$'s shares of the additive sharings of all the ciphertexts based on the secrets and $S_{j,2}, \ldots, S_{j,c}$'s shares. Sim_2 computes the output of $\mathcal{O}_2$ 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_2 doesn't follow the protocol to compute $S_{j,1}$'s share of $\langle \lambda_w \rangle$. Instead, Sim_2 computes it based on λ_w and the corrupted servers' 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_β in $\mathtt{Circ}^{V_j}$, Sim_2 doesn't follow the protocol to compute $\mathsf{ct}^{(i,1)}_{j_\beta,1\oplus s_{i,1}}$. Instead, Sim_2 samples a random κ-bit string as $\mathsf{ct}^{(i,1)}_{j_\beta,1\oplus s_{i,1}}$.

To prove that the distributions of $\mathbf{Hyb}_{20}$ and $\mathbf{Hyb}_{19}$ are computationally indistinguishable, we additionally construct the following hybrids between $\mathbf{Hyb}_{19}$ and $\mathbf{Hyb}_{20}$.

$\mathbf{Hyb}_{20.0}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 computes $S_{j,1}$'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, $\mathbf{Hyb}_{20.0}$ and $\mathbf{Hyb}_{19}$ have the same distribution.

$\mathbf{Hyb}_{20.1.1}$: In this hybrid, Sim_2 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 5.(a) and 5.(b) of the evaluation phase. Here the input of V_j, i.e. the output of $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ to the receiver is shared by an additive sharing among $S_{j,1}, \ldots, S_{j,c}$, and all the secrets of these sharings have been computed by Sim_2. 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 output of $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$, 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 output values of $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{Coin}}$ computed by Sim_2 is the same as the output of $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{Coin}}$ in Π_0. Since $V_j \in \mathcal{H}_{\mathsf{vir}}$, the corrupted parties just follow the protocol to distribute the sharings for the servers emulating V_j. From the correctness of the multiparty garbling process, the result $\overline{[\boldsymbol{s}_1]^{(2)}}$ is the same as $[\boldsymbol{s}_1]^{(2)}$ from the execution of Π_0.

In addition, if R_1 is an honest client, Sim_2 doesn't generate the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ first and then computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}]^{(3)}$ by himself. Instead, Sim_2 generates $[\boldsymbol{s}_1]^{(2)}$ first and then samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}]^{(3)}$ based on the shares for virtual servers out of $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}]^{(3)}$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}$ for each $\beta = 1, \ldots, c$. Then, Sim_2 computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}]^{(3)}, [\boldsymbol{s}_1]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$. Similarly, if R_1 is a virtual server in $\mathcal{H}_{\mathsf{vir}}$, Sim_2 doesn't generate the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},1}]^{(3)}$ first and then computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,1}]^{(3)}$ by himself. Instead, Sim_2 generates $[\boldsymbol{s}_1]^{(2)}$ first and then samples the whole sharing $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,1}]^{(3)}$ based on the shares for virtual servers out of $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,1}]^{(3)}$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,1}$. Then, Sim_2 computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},1}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,1}]^{(3)}, [\boldsymbol{s}_1]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},1} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},1}]$. Then for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 follows the computational process to compute each $Y^{S_{j,\beta}}_{a,s_{1,a}^{V_j}}$ in the protocol to compute each $Y^{S_{j,\beta}}_{a,s_{1,a}^{V_j}}$ with each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}]^{(3)}$ generated by the honest clients and the first servers emulating virtual servers in $\mathcal{H}_{\mathsf{vir}}$ being regarded as $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}_1]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$. For the other label $Y^{S_{j,\beta}}_{a,1\oplus s_{1,a}^{V_j}}$, Sim_2 still follows the protocol to compute it.

Since the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ generated by the honest clients and the first servers 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 secret in the last hybrid, their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}_1]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$ are also random based on the shares for virtual servers out of $\mathcal{H}_{\mathsf{vir}}$ and the secret. 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 $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ generated by an honest party without changing their distributions. Thus, $\mathbf{Hyb}_{20.1.1}$ and $\mathbf{Hyb}_{20.0}$ have the same distribution.

$\mathbf{Hyb}_{20.1.2}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 doesn't follow the protocol to compute the $S_{j,1}$'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_2 samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string as $k^{S_{j,1}}_{w,v_w\oplus\lambda_w}$. For each input wire of $\mathtt{Circ}^{V_j}$, Sim_2 samples a random $(\kappa-1)$-bit string as $k^{S_{j,1}}_{w,v_w\oplus\lambda_w}$. Then Sim_2 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_2 computes $k_{w,1\oplus v_w\oplus\lambda_w}^{S_{j,1}} = k_{w,v_w\oplus\lambda_w}^{S_{j,1}} \oplus \Delta^{S_{j,1}}$. 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, $\mathbf{Hyb}_{20.1.2}$ and $\mathbf{Hyb}_{20.1.1}$ have the same output distribution.

$\mathbf{Hyb}_{20.1.3}$: In this hybrid, Sim_2 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 $\{k_{a,v_a\oplus\lambda_a}^{S_{j,i}}, k_{b,v_b\oplus\lambda_b}^{S_{j,i}}\}_{i=1}^{c}$, Sim_2 checks whether each query of the honest server $S_{j,1}$ to the random oracle $\mathcal{O}_1$ has been queried before. If true, Sim_2 aborts the simulation. Otherwise, Sim_2 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 $\Delta^{S_{j,1}}$ are generated. Since each query made to the random oracle by $S_{j,1}$ contains either of the $(\kappa-1)$-bit strings $k_{a,1\oplus v_a\oplus\lambda_a}^{S_{j,1}}$ or $k_{b,1\oplus v_b\oplus\lambda_b}^{S_{j,1}}$ with $k_{a,1\oplus v_a\oplus\lambda_a}^{S_{j,1}} - k_{a,v_a\oplus\lambda_a}^{S_{j,1}} = k_{b,1\oplus v_b\oplus\lambda_b}^{S_{j,1}} - k_{b,v_b\oplus\lambda_b}^{S_{j,1}} = \Delta^{S_{j,1}}$ 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 $\mathbf{Hyb}_{20.1.3}$ and $\mathbf{Hyb}_{20.1.2}$ are computationally indistinguishable.

$\mathbf{Hyb}_{20.1.4}$: 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 $\{k_{a,v_a\oplus\lambda_a}^{S_{j,i}}, k_{b,v_b\oplus\lambda_b}^{S_{j,i}}\}_{i=1}^{c}$, Sim_2 samples random κ-bit strings as the ciphertexts and computes $S_{j,1}$'s shares of the additive sharings of the ciphertexts based on the secrets and $S_{j,2}, \ldots, S_{j,c}$'s shares. While emulating $\mathcal{O}_1$, 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 $\mathcal{O}_1(k_{a,i_0}^{S_{j,1}} \| i_0 \| k_{b,i_1}^{S_{j,1}} \| i_1 \| 1 \| j \| i \| g)$ for each $i = 1, \ldots, c$ based on the random strings and the wire labels of wire o. Note that the only difference between $\mathbf{Hyb}_{20.1.4}$ and $\mathbf{Hyb}_{20.1.3}$ 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_2 does not abort the simulation, queries in Q_1 have not been queried before. Thus, $\mathbf{Hyb}_{20.1.4}$ and $\mathbf{Hyb}_{20.1.3}$ have the same output distribution.

$\mathbf{Hyb}_{20.1.5}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 changes the order of sampling random κ-bit strings as the ciphertexts of this gate except those computed with $\{k_{a,v_a\oplus\lambda_a}^{S_{j,i}}, k_{b,v_b\oplus\lambda_b}^{S_{j,i}}\}_{i=1}^{c}$ and sampling $\Delta^{S_{j,1}}$ to decide the queries to $\mathcal{O}_1$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{20.1.5}$ and $\mathbf{Hyb}_{20.1.4}$ have the same output distribution.

$\mathbf{Hyb}_{20.1.6}$: In this hybrid, Sim_2 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 $S_{j,1}$'s share of $\langle \mathsf{ct}_{w,i_2}^{S_{j,\beta}} \rangle$ for each $\beta = 1, \ldots, c$, Sim_2 checks whether the query made by $S_{j,1}$ to the random oracle $\mathcal{O}_2$ has been queried before. If true, Sim_2 aborts the simulation. Otherwise, Sim_2 adds the query to Q_2. Note that each query made to the random oracle contains a $(\kappa-1)$-bit string $k_{w,1\oplus v_w\oplus\lambda_w}^{S_{j,1}}$ with $k_{w,1\oplus v_w\oplus\lambda_w}^{S_{j,1}} - k_{w,v_w\oplus\lambda_w}^{S_{j,1}} = \Delta^{S_{j,1}}$, for the same reason in $\mathbf{Hyb}_{20.1.3}$, the probability that some query has been queried (either by the honest server or by the adversary) is negligible. Thus, the distributions of $\mathbf{Hyb}_{20.1.6}$ and $\mathbf{Hyb}_{20.1.5}$ are computationally indistinguishable.

$\mathbf{Hyb}_{20.1.7}$: 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_2 samples a random $c\ell\kappa$-bit string as the ciphertext $\mathsf{ct}_{w,i_2}^{S_{j,\beta}}$ and computes $S_{j,1}$'s shares of the additive sharing of the ciphertext based on the secret and $S_{j,2}, \ldots, S_{j,c}$'s shares. While emulating $\mathcal{O}_2$, Sim_2 computes the output based on the random strings and the output labels. Note that the only difference between $\mathbf{Hyb}_{20.1.7}$ and $\mathbf{Hyb}_{20.1.6}$ is the way we decide the output for queries in Q_2. Since $\mathsf{ct}_{w,i_2}^{S_{j,\beta}}$ is randomly sampled, $\mathsf{ct}_{w,i_2}^{S_{j,\beta}} \oplus m$ is also uniformly random for any $c\ell\kappa$-bit string m. In particular, when Sim_2 does not abort the simulation, queries in Q_2 have not been queried before. Thus, $\mathbf{Hyb}_{20.1.7}$ and $\mathbf{Hyb}_{20.1.6}$ have the same output distribution.

$\mathbf{Hyb}_{20.1.8}$: 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_2 doesn't follow the protocol to compute $S_{j,1}$'s share of $\langle\lambda_w\rangle$. Instead, Sim_2 computes it based on λ_w and the corrupted servers' shares of $\langle\lambda_w\rangle$. Since $S_{j,1}$'s share 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 $S_{j,1}$'s share of $\langle\lambda_w\rangle$ is generated based on λ_w and the corrupted servers' 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 $S_{j,1}$'s share of $\langle\lambda_w\rangle$. Thus, $\mathbf{Hyb}_{20.1.8}$ and $\mathbf{Hyb}_{20.1.7}$ have the same output distribution.

$\mathbf{Hyb}_{20.1.9}$: In this hybrid, if the receiver R_1 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 the honest server $S_{j,1}$ computes $\mathsf{ct}^{(1,1)}_{j_\beta,1\oplus s_{1,\beta}}$, Sim_2 checks whether the query to the random oracle $\mathcal{O}_1$ has been queried before. If true, Sim_2 aborts the simulation. Otherwise, Sim_2 adds the query to Q_1. Since $\boldsymbol{r}_{1\oplus s_{1,\beta},\beta,1}$ is generated randomly in $\{0,1\}^\kappa$, the probability that some query has been queried (either by the honest server or by the adversary) is negligible. Thus, the distributions of $\mathbf{Hyb}_{20.1.9}$ and $\mathbf{Hyb}_{20.1.8}$ are computationally indistinguishable.

$\mathbf{Hyb}_{20.1.10}$: In this hybrid, if the receiver R_1 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_2 doesn't follow the protocol to compute $\mathsf{ct}^{(1,1)}_{j_\beta,1\oplus s_{1,\beta}}$. Instead, Sim_2 samples a random κ-bit string as $\mathsf{ct}^{(1,1)}_{j_\beta,1\oplus s_{1,\beta}}$. While emulating $\mathcal{O}_1$, Sim_2 computes the output of $\mathcal{O}_1(\boldsymbol{r}_{1\oplus s_{1,\beta},\beta}\|s_{1,\beta}\|1\|1\|\beta\|j_\beta)$ based on $\mathsf{ct}^{(1,1)}_{j_\beta,1\oplus s_{1,\beta}}$ and $k_{w_{j_\beta},1\oplus v_{w_{j_\beta}}^{S_{j,1}}\oplus\lambda_{w_{j_\beta}}}\|(1\oplus v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}})$. Note that the only difference between $\mathbf{Hyb}_{20.1.10}$ and $\mathbf{Hyb}_{20.1.9}$ is the way we decide the output for queries in Q_1. Since $\mathsf{ct}^{(1,1)}_{j_\beta,1\oplus s_{1,\beta}}$ is randomly sampled, $\mathsf{ct}^{(1,1)}_{j_\beta,1\oplus s_{1,\beta}}\oplus m$ is also uniformly random for any κ-bit string m. In particular, when Sim_2 does not abort the simulation, queries in Q_1 have not been queried before. Thus, $\mathbf{Hyb}_{20.1.10}$ and $\mathbf{Hyb}_{20.1.9}$ have the same output distribution.

$\mathbf{Hyb}_{20.1.11}$: In this hybrid, for virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 changes the order of sampling a random κ-bit string as the ciphertext $\mathsf{ct}^{(1,1)}_{j_\beta,1\oplus s_{1,\beta}}$ and sampling $\boldsymbol{r}_{1\oplus s_{1,\beta},\beta,1}$ to decide the queries to $\mathcal{O}_1$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{20.1.11}$ and $\mathbf{Hyb}_{20.1.10}$ have the same output distribution.

$\mathbf{Hyb}_{20.1.12}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 changes the order of sampling random $c\ell\kappa$-bit strings as the ciphertexts $\mathsf{ct}^{S_{j,\beta}}_{w,i_2}$ for each $\beta = 1,\ldots,c$ 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 $\Delta^{S_{j,1}}$ to decide the queries to $\mathcal{O}_2$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{20.1.12}$ and $\mathbf{Hyb}_{20.1.11}$ have the same output distribution.

Note that for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, $\Delta^{S_{j,1}}$ is not used before all the ciphertexts of the gates in $\mathtt{Circ}_1^{V_j}$ are generated. Sim_2 delays the generating of $\Delta^{S_{j,1}}$ after the garbling of $\mathtt{Circ}_1^{V_j}$ is completed.

Similarly, for each $\gamma = 2,\ldots,\mathsf{rec}$ we can define $\mathbf{Hyb}_{20.\gamma.1},\ldots,\mathbf{Hyb}_{20.\gamma.12}$.

$\mathbf{Hyb}_{20.\gamma.1}$: In this hybrid, Sim_2 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 5.(a) and 5.(b) of the evaluation phase. Here the input of V_j, i.e. the output of $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ to the receiver is shared by an additive sharing among $S_{j,1},\ldots,S_{j,c}$, and all the secrets of these sharings have been computed by Sim_2. 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 output of $\mathcal{F}_{\mathsf{prep}},\mathcal{F}_{\mathsf{input}}$ 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 $\mathtt{Circ}_\gamma^{S_j}$, for the same reason in $\mathbf{Hyb}_{20.1.1}$, the result $\overline{[\boldsymbol{s}_\gamma]^{(2)}}$ is the same as $[\boldsymbol{s}_\gamma]^{(2)}$ from the execution of Π_0.

In addition, if R_γ is an honest client, Sim_2 doesn't generate the honest virtual servers' shares of 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_2 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 out of $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}]^{(3)}$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}$ for each $\beta = 1,\ldots,c$. Then, Sim_2 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

R_γ is a virtual server in $\mathcal{H}_{\mathsf{vir}}$, Sim_2 doesn't generate the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},1}]^{(3)}$ first and then computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,1}]^{(3)}$ by himself. Instead, Sim_2 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 out of $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,1}]^{(3)}$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,1}$. Then, Sim_2 computes the shares for virtual servers in $\mathcal{H}_{\mathsf{vir}}$ of $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},1}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,1}]^{(3)}, [\boldsymbol{s}_\gamma]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},1} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},1}]$. Then for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 follows the computational process to compute each $Y^{S_{j,\beta}}_{a,s_{1,a}\oplus \boldsymbol{s}^{V_j}_{\gamma,a}}$ in the protocol to compute each $Y^{S_{j,\beta}}_{a,s_{1,a}}$ with each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_\gamma,\beta}]^{(3)}$ generated by the honest clients and the first servers emulating virtual servers in $\mathcal{H}_{\mathsf{vir}}$ regarded as $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)} + [\boldsymbol{s}_\gamma]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$. For the other label $Y^{S_{j,\beta}}_{a,1\oplus s^{V_j}_{\gamma,a}}$, Sim_2 still follows the protocol to compute it. For the same reason in $\mathbf{Hyb}_{20.1.1}$, $\mathbf{Hyb}_{20.\gamma.1}$ and $\mathbf{Hyb}_{20.(\gamma-1).12}$ have the same distribution.

$\mathbf{Hyb}_{20.\gamma.2}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 doesn't follow the protocol to compute the $S_{j,1}$'s shares of the garbled circuit for $\mathtt{Circ}^{V_j}_\gamma$. Instead, for each wire w in $\mathtt{Circ}^{V_j}_\gamma$ that is not an input wire of the circuit and is not an output wire of an XOR gate or an output gate, Sim_2 samples a random bit as $v_w \oplus \lambda_w$ and a random $(\kappa-1)$-bit string as $k^{S_{j,1}}_{w,v_w\oplus\lambda_w}$. Then Sim_2 computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$ for these wires. For each wire w in $\mathtt{Circ}^{V_j}_\gamma$ that is not an output wire of an output gate, Sim_2 computes $k^{S_{j,1}}_{w,1\oplus v_w\oplus\lambda_w} = k^{S_{j,1}}_{w,v_w\oplus\lambda_w} \oplus \Delta^{S_{j,1}}$. For the same reason in $\mathbf{Hyb}_{20.1.2}$, $\mathbf{Hyb}_{20.\gamma.2}$ and $\mathbf{Hyb}_{20.\gamma.1}$ 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 $\mathtt{Circ}^{V_j}_\gamma$ with input wire a, b, and for the ciphertexts of this gate except those computed with $\{k^{S_{j,i}}_{a,v_a\oplus\lambda_a}, k^{S_{j,i}}_{b,v_b\oplus\lambda_b}\}^c_{i=1}$, Sim_2 checks whether each query of the honest server $S_{j,1}$ to the random oracle $\mathcal{O}_1$ has been queried before. If true, Sim_2 aborts the simulation. Otherwise, Sim_2 adds the query to Q_1. For the same reason in $\mathbf{Hyb}_{20.1.3}$, the distributions of $\mathbf{Hyb}_{20.\gamma.3}$ and $\mathbf{Hyb}_{20.\gamma.2}$ 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 $\mathtt{Circ}^{V_j}_\gamma$ with input wire a, b and output wire o, and for the ciphertexts of this gate except those computed with $\{k^{S_{j,i}}_{a,v_a\oplus\lambda_a}, k^{S_{j,i}}_{b,v_b\oplus\lambda_b}\}^c_{i=1}$, Sim_2 samples random κ-bit strings as the ciphertexts and computes $S_{j,1}$'s shares of the additive sharings of the ciphertexts based on the secret and $S_{j,2}, \ldots, S_{j,c}$'s shares. While emulating $\mathcal{O}_1$, 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 $\mathcal{O}_1(k^{S_{j,1}}_{a,i_0} \| i_0 \| k^{S_{j,1}}_{b,i_1} \| i_1 \| 1 \| j \| i \| g)$ for each $i = 1, \ldots, c$ based on the random strings and the wire labels of wire o. For the same reason in $\mathbf{Hyb}_{20.1.4}$, $\mathbf{Hyb}_{20.\gamma.4}$ and $\mathbf{Hyb}_{20.\gamma.3}$ have the same output distribution.

$\mathbf{Hyb}_{20.\gamma.5}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 changes the order of sampling random κ-bit strings as the ciphertexts of this gate except those computed with $\{k^{S_{j,i}}_{a,v_a\oplus\lambda_a}, k^{S_{j,i}}_{b,v_b\oplus\lambda_b}\}^c_{i=1}$ and sampling $\Delta^{S_{j,1}}$ to decide the queries to $\mathcal{O}_1$. For the same reason in $\mathbf{Hyb}_{20.1.5}$, $\mathbf{Hyb}_{20.\gamma.5}$ and $\mathbf{Hyb}_{20.\gamma.4}$ have the same output distribution.

$\mathbf{Hyb}_{20.\gamma.6}$: 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 $S_{j,1}$'s share of $\langle \mathsf{ct}^{S_{j,\beta}}_{w,i_2} \rangle$ for each $\beta = 1, \ldots, c$, Sim_2 checks whether the query made by $S_{j,1}$ to the random oracle $\mathcal{O}_2$ has been queried before. If true, Sim_2 aborts the simulation. Otherwise, Sim_2 adds the query to Q_2. For the same reason in $\mathbf{Hyb}_{20.1.6}$, the distributions of $\mathbf{Hyb}_{20.\gamma.6}$ and $\mathbf{Hyb}_{20.\gamma.5}$ are computationally indistinguishable.

$\mathbf{Hyb}_{20.\gamma.7}$: 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_2 samples a random $c\ell\kappa$-bit string as the ciphertext $\mathsf{ct}^{S_{j,\beta}}_{w,i_2}$ and computes $S_{j,1}$'s shares of the additive sharing of the ciphertext based on the secret and $S_{j,2}, \ldots, S_{j,c}$'s shares. While emulating $\mathcal{O}_2$, Sim_2 computes the output based on the random strings and the output labels. For the same reason in $\mathbf{Hyb}_{20.1.7}$, $\mathbf{Hyb}_{20.\gamma.7}$ and $\mathbf{Hyb}_{20.\gamma.6}$ have the same output distribution.

$\mathbf{Hyb}_{20.\gamma.8}$: 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_2 doesn't follow the protocol to compute $S_{j,1}$'s share of $\langle \lambda_w \rangle$. Instead, Sim_2 computes it based

on λ_w and the corrupted servers' shares of $\langle\lambda_w\rangle$. For the same reason in $\mathbf{Hyb}_{20.1.8}$, $\mathbf{Hyb}_{20.\gamma.8}$ and $\mathbf{Hyb}_{20.\gamma.7}$ have the same output distribution.

$\mathbf{Hyb}_{20.\gamma.9}$: In this hybrid, if the receiver R_γ 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 the honest server $S_{j,1}$ computes $\mathsf{ct}^{(\gamma,1)}_{j_\beta,1\oplus s_{\gamma,\beta}}$, Sim_2 checks whether the query to the random oracle $\mathcal{O}_1$ has been queried before. If true, Sim_2 aborts the simulation. Otherwise, Sim_2 adds the query to Q_1. For the same reason in $\mathbf{Hyb}_{20.1.9}$, the distributions of $\mathbf{Hyb}_{20.\gamma.9}$ and $\mathbf{Hyb}_{20.\gamma.8}$ are computationally indistinguishable.

$\mathbf{Hyb}_{20.\gamma.10}$: In this hybrid, if the receiver R_γ 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_2 doesn't follow the protocol to compute $\mathsf{ct}^{(\gamma,1)}_{j_\beta,1\oplus s_{\gamma,\beta}}$. Instead, Sim_2 samples a random κ-bit string as $\mathsf{ct}^{(\gamma,1)}_{j_\beta,1\oplus s_{\gamma,\beta}}$. While emulating $\mathcal{O}_1$, Sim_2 computes the output of $\mathcal{O}_1(\boldsymbol{r}_{1\oplus s_{\gamma,\beta},\beta}\|s_{\gamma,\beta}\|\gamma\|1\|\beta\|j_\beta)$ based on $\mathsf{ct}^{(\gamma,1)}_{j_\beta,1\oplus s_{\gamma,\beta}}$ and $k_{w_{j_\beta},1\oplus v^{S_{j,1}}_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}}}\|(1\oplus v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}})$. For the same reason in $\mathbf{Hyb}_{20.1.10}$, $\mathbf{Hyb}_{20.\gamma.10}$ and $\mathbf{Hyb}_{20.\gamma.9}$ have the same output distribution.

$\mathbf{Hyb}_{20.\gamma.11}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 changes the order of sampling a random κ-bit string as the ciphertext $\mathsf{ct}^{(\gamma,1)}_{j_\beta,1\oplus s_{\gamma,\beta}}$ and sampling $\boldsymbol{r}_{1\oplus s_{\gamma,\beta},\beta,1}$ to decide the queries to $\mathcal{O}_1$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{20.\gamma.11}$ and $\mathbf{Hyb}_{20.\gamma.10}$ have the same output distribution.

$\mathbf{Hyb}_{20.\gamma.12}$: In this hybrid, for each virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, Sim_2 changes the order of sampling random $c\ell\kappa$-bit strings as the ciphertexts $\mathsf{ct}^{S_{j,\beta}}_{w,i_2}$ for each $\beta = 1,\ldots,c$ 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 $\Delta^{S_{j,1}}$ to decide the queries to $\mathcal{O}_2$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{20.\gamma.12}$ and $\mathbf{Hyb}_{20.\gamma.11}$ have the same output distribution.

Note that $\mathbf{Hyb}_{20.\mathsf{rec}.12}$ is just $\mathbf{Hyb}_{20}$, we conclude that the distributions of $\mathbf{Hyb}_{20}$ and $\mathbf{Hyb}_{19}$ are computationally indistinguishable.

Note that for each honest virtual server $V_j \in \mathcal{H}_{\mathsf{Vir}}$, $\Delta^{S_{j,1}}$ is not used before all the ciphertexts of the gates in $\mathtt{Circ}^{V_j}_\gamma$ are generated. Sim_2 delays the generating of $\Delta^{S_{j,1}}$ after the garbling of $\mathtt{Circ}^{V_j}_\gamma$ is completed.

Also note that if R_i is an honest client, $\boldsymbol{r}^{(\alpha)}_{\mathbf{1}\oplus\boldsymbol{s}_i,\beta}$ for $\alpha = 1,\ldots,\kappa$ and $\beta = 1,\ldots,c$ are not used until the whole garbled circuit of V_j is generated. Sim_2 generates them after the whole garbled circuit is generated in future hybrids to decide the set Q_1.

$\mathbf{Hyb}_{21}$: 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_2 doesn't compute the output of $\mathcal{O}_1$ to the queries that are used to generate these ciphertexts based on the random strings and the wire labels of wire o. For each output gate in $\mathtt{Circ}^{V_j}$ indexed $1,\ldots,f.m$ with input wire w, Sim_2 doesn't compute the output of $\mathcal{O}_2$ to the queries that are used to generate these ciphertexts based on the random strings and the output labels. Instead, Sim_2 honestly emulates the random oracles. In particular, Sim_2 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 $\mathbf{Hyb}_{21}$ and $\mathbf{Hyb}_{20}$ are computationally indistinguishable.

For the sake of contradiction, assume that there exists an adversary $\mathcal{A}_1$ such that $\mathbf{Hyb}_{21}$ and $\mathbf{Hyb}_{20}$ are computationally distinguishable. Let Q_1, Q_2 be the set of queries to the random oracles $\mathcal{O}_1, \mathcal{O}_2$ respectively when Sim_2 is computing the cipher-texts 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 $\mathbf{Hyb}_{20}$ and $\mathbf{Hyb}_{21}$ 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 $\mathbf{Hyb}_{21}$ and $\mathbf{Hyb}_{20}$ 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 $\mathbf{Hyb}_{21}$. 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 $k^{S_{j,1}}_{w,1\oplus v_w\oplus\lambda_w}$ for a wire

w in an honest server S_j's circuit $\mathtt{Circ}^{V_j}$. Since $\Delta^{S_{j,1}}$ 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 $\Delta^{S_{j,1}}$. Therefore, $k^{S_{j,1}}_{w,1\oplus v_w\oplus\lambda_w} = k^{S_{j,1}}_{w,v_w\oplus\lambda_w} \oplus \Delta^{S_{j,1}}$ 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 $R_i \in \mathcal{H}_{\mathsf{vir}}$, since $\boldsymbol{r}_{1\oplus s_{i,\eta},\eta,1}$ 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 $\mathbf{Hyb}_{21}$ and $\mathbf{Hyb}_{20}$ are computationally indistinguishable.

Note that for virtual server $V_j \in \mathcal{H}_{\mathsf{vir}}$, $\Delta^{S_{j,1}}$ is not used in the simulation, Sim_2 doesn't generate it in future hybrids.

Also note that if R_i is an honest client, $\boldsymbol{r}^{(\alpha)}_{\mathbf{1}\oplus\boldsymbol{s}_i,\beta}$ for $\alpha = 1,\dots,\kappa$ and $\beta = 1,\dots,c$ are not used until R_i receives $\boldsymbol{s}_i, \{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}\}^{\kappa}_{\alpha=1}$ for each $\beta = 1,\dots,c$ and does the verification on them. Sim_2 delays the generation of $\{\boldsymbol{r}^{(\alpha)}_{\mathbf{1}\oplus\boldsymbol{s}_i,\beta}\}^{\kappa}_{\alpha=1}$ for each $\beta = 1,\dots,c$ after R_i receives $\boldsymbol{s}_i, \{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}\}^{\kappa}_{\alpha=1}$ for each $\beta = 1,\dots,c$ in future hybrids. If R_i is a virtual server in $\mathcal{H}_{\mathsf{vir}}$, Sim_2 doesn't need $\{\boldsymbol{r}^{(\alpha)}_{\mathbf{1}\oplus\boldsymbol{s}_i,1}\}^{\kappa}_{\alpha=1}$ in the simulation, so Sim_2 does not generate them in future hybrids. Now all the shares for $S_{j,1}$ with $V_j \in \mathcal{H}_{\mathsf{vir}}$ generated by an honest party in the sharing phase are not used in the simulation, Sim_2 doesn't generate them in future hybrids.

$\mathbf{Hyb}_{22}$: In this hybrid, for each $i = 1,\dots,\mathsf{rec}$ with honest receiver R_i who is a client, Sim_2 doesn't follow the protocol to check the values $\boldsymbol{s}_i$ and $\{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}\}^{\kappa}_{\alpha=1}$ for $\beta = 1,\dots,c$ received from P_{king}. Instead, Sim_2 checks whether they are correctly sent. Note that when they are correctly sent, then $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta} = \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta} + \boldsymbol{s}_i * (\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta})$ must hold for each $\alpha = 1,\dots,\kappa$ and $\beta = 1,\dots,c$. Thus, the output only changes when $\boldsymbol{s}_i, \{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}\}^{\kappa}_{\alpha=1}$ are not correctly sent but for each $\alpha = 1,\dots,\kappa$ it still holds that $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i} = \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta} + \boldsymbol{s}_i * (\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta})$ (by the values received from P_{king}). Since when $\boldsymbol{s}_i$ is correctly sent, $\{\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}\}^{\kappa}_{\alpha=1}$ is determined by the equation $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i} = \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta} + \boldsymbol{s}_i * (\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta})$, the output only changes when a bit of $\boldsymbol{s}_i$ is not correctly sent. Assume that it's the η-th bit. Note that if the η-th bit of $\boldsymbol{s}_i$ is 0, then the η-th bit of $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$ is the η-th bit of $\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta}$, which is sampled randomly after $\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}, \boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta}$ are received from P_{king}. Similarly, if the η-th bit of $\boldsymbol{s}_i$ is 1, then the η-th bit of $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$ is the η-th bit of $\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}$, which is also sampled randomly. Thus, the output changes only when κ randomly sampled bits are all guessed correctly by $\mathcal{A}$. The probability is $2^{-\kappa}$, which is negligible. Thus, the distributions of $\mathbf{Hyb}_{22}$ and $\mathbf{Hyb}_{21}$ are statistically close.

Note that if R_i 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 $\beta = 1,\dots,c$ for the simulation, and we also don't need honest servers' shares of $\{[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]\}^{\kappa}_{\alpha=1}$ for $\beta = 1,\dots,c$ in the simulation. Sim_2 doesn't generate them in future hybrids.

$\mathbf{Hyb}_{23}$: In this hybrid, since all the transcripts between honest and corrupted parties generated by Sim_2 can be generated from the transcripts between honest and corrupted parties obtained in the execution of Π_0, Sim_2 just runs Π_0 first to obtain all the transcripts and then uses them to generate the output of Sim_2. In addition, honest clients don't follow the protocol Π_2 to compute their output. Instead, they follow Π_0 to get their output. Since the value $\boldsymbol{s}_i$ sent from P_{king} to each honest client in Π_2 is the same as the value $\boldsymbol{s}_i$ in Π_0, and the preprocessing and input data of Π_0, Π_2 is also the same, the computation of honest clients' outputs in the two protocols is completely the same. Therefore, we only change the way of generating the output of Sim_2 without changing their distributions. Thus, $\mathbf{Hyb}_{23}$ and $\mathbf{Hyb}_{22}$ have the same distribution.

$\mathbf{Hyb}_{24}$: In this hybrid, Sim_2 doesn't run Π_0 to get the transcripts between honest and corrupted parties in Π_0 and use them to all the transcripts between honest and corrupted parties in Π_2. Instead, note that each output of $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{Coin}}$ to the receiver (which is a virtual server, say V_j) is shared by an additive sharing among $S_{j,1},\dots,S_{j,c}$, and all the secrets of these sharings have been computed by Sim_2. We can regard the secrets as the messages sent between honest and corrupted parties in Π_0. In this way, Sim_2 constructs an adversary A' in Π_0 that interacts with honest parties in Π_0 with the secrets of these additive sharings while interacting with $\mathcal{F}_{\mathsf{prep}}, \mathcal{F}_{\mathsf{Coin}}$ and then follows the protocol in the evaluation phase of Π_0. Sim_2 invokes Sim_0 with $\mathcal{A}'$ to get the transcripts between honest and corrupted parties in Π_0. In addition, honest clients get their outputs from $\mathcal{F}$ instead of following Π_0 to compute them. From the requirements of Π_0, the joint

distribution of transcripts between honest and corrupted parties in Π_0 and the honest clients' output in Π_0 is computationally indistinguishable from the joint distribution of the output of Sim_0 and honest clients' output from $\mathcal{F}$. Thus, the distributions of $\mathbf{Hyb}_{24}$ and $\mathbf{Hyb}_{23}$ are computationally indistinguishable.

Note that $\mathbf{Hyb}_{24}$ is the ideal-world scenario, Π_2 computes $\mathcal{F}$ with computational security.

□

J.4 Cost Analysis for Π_2

J.4.1 Analysis of Communication Complexity

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

Determining the Virtual Servers: This step only contains interaction with $\mathcal{F}_{\mathsf{Coin}}$. The instantiation of $\mathcal{F}_{\mathsf{Coin}}$ requires communication of $O((m+n)^2\kappa)$ bits.

Emulating $\mathcal{F}_{\mathsf{prep}}$:

1. **Transforming the Circuit.** This step only contains local computation and requires no communication.
2. **Preparing Random Sharings.** In this step, each server generates $10WN\ell/(ank^2\epsilon) = O(|C'|/(nN\log n))$ random $\Sigma^{(2)}_{\times a}$-sharings, where each share is additively distributed to c servers. The size of each sharing is $caN\ell^2 = O(N\log n)$. Thus, the total cost is $O(|C'|)$.
3. **Preparing Zero Sharings.** In this step, each server generates $10WN/(ank^2\epsilon) + W_O/(ank\epsilon) = O(|C'|/(nN\log n))$ random $\Sigma^{(2)}_{\times a}$-sharings with all-zero secrets, where each share is additively distributed to c servers. The size of each sharing is $O(N\log n)$. Thus, the total cost is $O(|C'|)$.
4. **Preparing Masks for Output Sharings.** In this step, the clients need to send a Σ-sharing of size $O(N)$ for each batch of $k = O(N)$ output wires, where each share is additively distributed to c servers. The communication is linear to the number W_I of input wires of C', i.e. $O(|W_I|) < O(|C'|)$ bits.
5. **Preparing Masks for Transpose Protocols.** In this step, the servers distributes $10cWN/k^2 = O(|C'|/N)$ $\Sigma^{(2)}$-sharings of size $O(N)$, where each share is additively distributed to c servers. Thus, the communication cost of this step is $O(|C'|)$.
6. **Preprocessing for the Verification of Sharings.** In this step, the servers distributes n $\Sigma_{\times\kappa'}$-sharings and $2n$ $\Sigma^{(2)}_{\times\kappa'}$-sharings of size $O(N\kappa')$, where each share is additively distributed to c servers. Thus, the communication cost of this step is $O(nN\kappa')$.

Emulating $\mathcal{F}_{\mathsf{input}}$:

- In this step, for each group of $k = O(N)$ input wires of C', a client needs to distribute a Σ-sharing of $O(N)$ bits, where each share is additively distributed to c servers. Thus, the communication cost of this step is $O(|W_I|) < O(|C'|)$ bits.

Preparing for the Garbling of Local Circuits:

1. **Calling $\mathcal{F}_{\mathsf{ROT}}$.** This step only contains interaction with $\mathcal{F}_{\mathsf{ROT}}$. The total number of instances of ROTs is $c^2 \cdot ((4c+1)G_A + 2Nc\ell^2\mathsf{rec}) = O(|C'|)$.
2. **Preparing for the Output Labels.** In this step, for each $i = 1, \ldots, \mathsf{rec}$, each receiver of a $\Sigma^{(2)}$-sharing in Π_0 needs to distribute $c\kappa$ $\Sigma^{(3)}$-sharings and $c\kappa$ Σ-sharings, where each share is additively distributed to c servers. Thus, the communication cost of this step is $O(c\ell\kappa \cdot \mathsf{CC}^{\Pi_0}_{\mathsf{eval}}) = O(\mathsf{CC}^{\Pi_0}_{\mathsf{eval}})$, where $\mathsf{CC}^{\Pi_0}_{\mathsf{eval}}$ is the communication cost of the evaluation phase of Π_0 running by m clients and N servers.
3. **Generating Local Randomness.** This step only contains local computation and requires no communication.

Committing Local Inputs:

- This step only contains interaction with $\mathcal{F}_{\mathsf{Commit}}$. Each commitment is of length $O(\kappa)$. The total number of invocations to $\mathcal{F}_{\mathsf{Commit}}$ is cN, and they can be done in parallel. Thus, sending them to all the servers and letting them cross-check on them leads to $O(nN\kappa)$ bits of communication.

Local Computation Phase. This phase only contains local computation of real-world servers and requires no communication.

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

1. **Computing Output Labels.** This step only contains local computation and requires no communication.
2. **Garbling Local Circuits.** 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} $O(|C'|)$ times, resulting in a total communication of $O(|C'|\kappa)$ bits.
3. **Masking Input Wire Values.** In this step, the servers needs to open DS bits of values (recall that DS is the size of outputs from $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{input}}$ in Π_0). Opening each bit requires $c^2 = O(1)$ bits of communication. Thus, the total communication of this step is $O(\mathsf{DS})$.

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

1. **Verification of the Sharings.** In this step, the servers first need to call $\mathcal{F}_{\mathsf{Coin}}$, which requires $O(n^2\kappa')$-bit communication. Then the servers need to send an additive sharing of each virtual server V_α's share of a $\Sigma_{\times\kappa'}$-sharing and two $\Sigma^{(2)}_{\times\kappa'}$-sharings to all the servers, which needs $n(\ell + 2\ell^2) \cdot Nc\kappa' = O(nN\kappa')$ for all the servers. In addition, each server needs to send a result from $\mathcal{O}_1$ to all the servers, which requires $O(n^2\kappa)$ bits of communication. Thus, the total communication cost of this step is $O(nN\kappa')$.
2. **Verification of Local Computation.** In this step, each server needs to send a set of $N/16n$ virtual servers to all the servers and let them do a cross-check on the set, which requires $O(n^2 \cdot (N/16n) \cdot \log N) = O(nN \log N)$ bits of communication. Besides, for each server S, the communication messages during the local computation of virtual servers in Ver^S need to be sent to S, which requires communication of $O(|C'|\kappa/n)$ bits. The committed input of $S_{j,1}, \ldots, S_{j,c}$ to the local computation of each $V_j \in \mathsf{Ver}^S$ should also be opened to S, which requires $O(\mathsf{DS}/n)$ bits. To sum up, the total communication cost of this step is $O(|C'|\kappa + \mathsf{DS} + nN \log N)$.

Evaluation Phase. We analyze the communication cost of the evaluation phase step by step as follows:

1. **Sending Output Masks.** In this step, for each output wire of a virtual server's local circuit, the servers emulating the virtual server need to send a bit to P_{king}. The total number of output wires is $O(\mathsf{CC}^{\Pi_0}_{\mathsf{eval}})$. Thus, the communication of this step is $O(\mathsf{CC}^{\Pi_0}_{\mathsf{eval}})$.
2. **Encrypting Input Labels.** In this step, for each reconstruction of $\Sigma^{(2)}$-sharings to a virtual server receiver, the virtual server needs to send $2ck = O(N)$ ciphertexts of $O(\kappa)$ bits to P_{king}, resulting in communication of $O(\kappa \cdot \mathsf{CC}^{\Pi_0}_{\mathsf{eval}})$ bits.
3. **Sending Input Labels.** In this step, for each input wire of a virtual server's local circuit whose value does not come from reconstructions, the servers emulating the virtual server need to send κ bits to P_{king}. The total number of such input wires is DS. Thus, the total communication cost of this step is $O(\mathsf{DS} \cdot \kappa)$.

4. **Sending Garbled Circuits.** In this step, the virtual servers send their garbled circuits to P_{king}. Similar as in Π_1', the total garbled circuit size is $O((\mathsf{DS}+G_A)\cdot\kappa+\kappa\cdot\mathsf{CC}_{\mathsf{eval}}^{\Pi_0})$. Each element of the garbled circuits is shared among c servers, which leads to a $c=O(1)$ multiplication overheads. Thus, the communication of this step is still $O((\mathsf{DS}+G_A)\cdot\kappa+\kappa\cdot\mathsf{CC}_{\mathsf{eval}}^{\Pi_0})$.

5. **Evaluating the Circuit.** This step only contains local computation and requires no communication.

6. **Sending Outputs.** In this step, the communication comes from $c\kappa$ secrets of $\Sigma^{(3)}$-sharings and a secret of a $\Sigma^{(2)}$-sharing for each reconstruction of a $\Sigma^{(2)}$-sharing with client receiver in the evaluation phase of Π_0. The size of $c\kappa$ secrets of $\Sigma^{(3)}$-sharings and a secret of a $\Sigma^{(2)}$-sharing is $O(k\kappa)=O(N\kappa)$ while the size of a $\Sigma^{(2)}$-sharing is $O(N\ell^2)$. Thus, the communication cost of this step is $O(\kappa\mathsf{CC}_{\mathsf{eval}}^{\Pi_0})$.

Taking $\mathsf{CC}_{\mathsf{eval}}^{\Pi_0}=O(|C'|)=O(|C|+DN^2+mN)$ and $\mathsf{DS}=O(|C'|)$ from the analyze of Π_0, and taking $N=O(n+\kappa),\kappa'=N+\kappa$, the total communication cost CC^{Π_2} of Π_2 is $O(|C|\kappa+D(n+\kappa)^2\kappa+m(n+\kappa)\kappa+n^3+m^2\kappa)$ plus $O(|C|+D(n+\kappa)^2+m(n+\kappa))$ instances of ROT of message length $O(\kappa)$.

J.4.2 Analysis of Rounds

We analyze the number of rounds we need in Π_2 as follows:

Sharing Phase. All the parties first need 2 rounds to agree on the random coin to determine the virtual servers. Then, all the sharings can be generated in parallel, and the servers can also call $\mathcal{F}_{\mathsf{ROT}}$ after the virtual servers are determined. After getting all the shares and all outputs from $\mathcal{F}_{\mathsf{ROT}}$, the servers need another 2 rounds to send their commitments and cross-check on them. Thus, the sharing phase requires $R^{\mathsf{ROT}}+4$ rounds, where R^{ROT} is the number of rounds needed for an instance of random OT with message length $\kappa-1$.

Local Computation Phase. This phase only contains local computation of real-world servers and requires no communication rounds.

Garbling Phase. In the garbling phase, the servers emulating each virtual server need to jointly compute 2 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 garbling phase requires 2 rounds.

Verification Phase. In the verification phase, the servers first need to agree on a random coin, which requires 2 rounds of communication, where the first round can be performed in parallel with the sharing phase. Then, the servers emulating each virtual server should send the secrets of three additive sharings to all the servers, which requires one round. After that, the servers use one round to cross-check the sharings. Thus, the verification of sharings requires 3 rounds in total. Note that this step can be performed in parallel with the garbling phase, we only need one additional round. Then, the servers need to verify the local computations of the virtual servers. Each server should first send a set and then let the servers cross-check on it, which needs two rounds. Then, the opening of commitments, the messages during the local computation of local servers, and the shares generated in the sharing phase can be sent together in one round. Thus, the verification phase requires 4 additional rounds in total.

Evaluation Phase. In the evaluation phase, the output masks, the ciphertexts, the input labels, and the garbled circuits can be sent in parallel (i.e. Steps 1-4 can be done in one round). Then, another round is required to let the evaluator send outputs to the clients. Thus, the evaluation phase requires 2 rounds.

To sum up, the total number of rounds we need in Π_2 is $12 + R^{\mathsf{ROT}}$, where R^{ROT} is the number of rounds needed for an instance of random OT with message length $\kappa - 1$.

K Corollary in the Standard Honest Majority Setting

Note that we may view the standard honest majority setting as a special case of the dishonest majority setting. Then, to remove the assumption of oblivious transfers in Theorem 2, we utilize the technique of [**?**] that makes use of a generic honest majority MPC protocol to help each pair of parties prepare a small number of OTs so that these two parties may use the OT extension technique [IKNP03, KOS15] to generate a sufficient number of ROTs.

We first give the functionality $\mathcal{F}_{\mathsf{OT}}$ of the OTs.

Functionality $\mathcal{F}_{\mathsf{OT}}(\kappa, \ell)$

The trusted party interacts with two parties P_1, P_2.

1. For each $i = 1, \ldots, \ell$, the trusted party receives two messages $v_{0,i}, v_{1,i} \in \{0,1\}^\kappa$ from P_1 and κ bits $x_i \in \{0,1\}$ from P_2.
2. For each $i = 1, \ldots, \ell$, the trusted party sends $v_{x_i,i}$ to P_2.

Figure 39: Functionality for ℓ oblivious transfers.

We can instantiate $\mathcal{F}_{\mathsf{OT}}(\kappa, \kappa)$ by using the DN protocol with malicious security in the honest majority setting [CGH+18] with information-theoretic security. We sketch the protocol as follows.

1. For each $i = 1, \ldots, \kappa$, P_2 shares its choice bit x_i by a degree-t Shamir secret sharing as $[x_i]_t$ to all parties.
2. For each $i = 1, \ldots, \kappa$, all parties invoke DN protocol to compute their shares of $[x_i \cdot (x_i - 1)]_t$ and reconstruct the output to P_1 who checks whether the reconstructed result equals 0. If not, P_1 aborts the protocol.
3. For each $i = 1, \ldots, \kappa$, P_1 secret shares its two messages $v_{0,i}, v_{1,i}$ as $[v_{0,i}]_t, [v_{1,i}]_t$ to all parties.
4. For each $i = 1, \ldots, \kappa$, all the parties invoke DN protocol to compute their shares of $[z_i]_t$ with $z_i = v_{0,i} \cdot (1 - x_i) + v_{1,i} \cdot x_i$.
5. For each $i = 1, \ldots, \kappa$, all the parties send their shares of $[z_i]_t$ to P_2 for reconstruction.

The communication complexity of this part is $O(n\kappa^2 + n^2\kappa)$ bits and the number of rounds is 13 for every pair of two parties assuming we use DN protocol with malicious security in [CGH+18].

With $\mathcal{F}_{\mathsf{OT}}(\kappa, \kappa)$ as base-OTs at hand, we take advantage of the result in [KOS15] which extends κ OT correlations to any number of OT correlations with the same message length between two parties with malicious security. The communication complexity of realizing $\mathcal{F}_{\mathsf{OT}}(\kappa, \ell)$ between two parties is $O(\ell\kappa)$ bits plus one invocation of $\mathcal{F}_{\mathsf{OT}}(\kappa, \kappa)$ and the number of rounds required is 3 plus the number rounds needed for realizing $\mathcal{F}_{\mathsf{OT}}(\kappa, \kappa)$.

Note that for ℓ instances of $\mathcal{F}_{\mathsf{ROT}}$ between P_1 and P_2, we only need to ask P_1 to send random vectors and ask P_2 to send random bits to $\mathcal{F}_{\mathsf{OT}}(\kappa, \ell)$. In the above way, to realize the $O(|C| + D(n+\kappa)^2)$ instances of $\mathcal{F}_{\mathsf{ROT}}$ with message length $O(\kappa)$ required in Π_2, we need communication of $O(|C|\kappa + D(n+\kappa)^2\kappa)$ bits and the number of rounds is $R^{\mathsf{ROT}} = 16$. As a result, we obtain the following corollary.

Corollary 1. *Assuming random oracles, there exists a computationally secure 28-round MPC protocol against a fully malicious adversary controlling up to $(n-1)/2$ parties with communication of $O(|C|\kappa + D(n+\kappa)^2\kappa + n^3)$ bits, where $|C|$ is the circuit size, D is the circuit depth, and κ is the computational security parameter.*

Remark 4. *The protocol in [BGH+23] requires a round complexity of 31 plus a constant parameter that defines the error distribution of LPN in the strong honest majority setting. Compared to them, we achieve a better round complexity.*