Achieving Guaranteed Output Delivery MPC with Constant Rounds and Linear Communication in Minicrypt

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

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

Abstract

In this work, we study the communication complexity of constant-round MPC with guaranteed output delivery (GOD) in Minicrypt. We construct the first MPC protocol in this setting with linear communication complexity of $O(|C|n\kappa + Dn^3\kappa^3 + W_I\mathsf{poly}(n,\kappa))$ bits under the assumption of a random oracle, where $|C|$ is the circuit size, D is the circuit depth, W_I is the number of input wires, and κ is the security parameter.

In comparison, the previously best-known construction with linear communication ($O(|C|n)$), presented by Goyal et al. (CRYPTO 2020), requires $O(D + n^2)$ round complexity. When targeting $O(D)$ round complexity, the best-known result by Agarwal et al. (ASIACRYPT 2024) still requires $O(|C|n^3)$ communication complexity. More communication is needed to achieve constant round complexity, even with non-black-box use of the underlying cryptographic primitives.

Contents

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

2 Technical Overview **6**
- 2.1 Background: Constant-Size Multiparty Garbled Circuit . . . 6
- 2.2 Augmenting the Verification with Identifiable Abort . . . 8
- 2.3 Handling Corrupted Parties Identified in the Verification . . . 9
- 2.4 Reconstruction of Garbled Circuits to Evaluators . . . 10
- 2.5 Summary of Our Protocol . . . 11

3 Preliminaries **12**
- 3.1 Secret Sharing . . . 12
- 3.2 Basic Subprotocols . . . 13

4 The Structure of the Underlying Multiparty Garbling Protocol **14**
- 4.1 The Preprocessing Phase . . . 14
- 4.2 The Garbling Phase . . . 16
- 4.3 The Verification Phase . . . 16
- 4.4 The Evaluation Phase . . . 18

5 Verifiable P2P Channels **19**

6 Compiler for Achieving GOD **20**
- 6.1 High-Level Structure . . . 20
- 6.2 Compiling the Preprocessing . . . 20
- 6.3 Verification of the Sharings . . . 21
- 6.4 Compiling the Garbling Phase . . . 22
- 6.5 Verification of Local Computations . . . 23
- 6.6 Collecting the Outputs from the Garbling Phase . . . 23

7 Main Protocol **24**

A An Overview for [GLOS25] **31**

B Additional Preliminaries **32**
- B.1 Security Model . . . 32
- B.2 Linear Secret Sharing Schemes. . . . 33
- B.3 Verifiable Secret Sharings . . . 34
- B.4 The Functionality $\mathcal{F}_{\mathsf{MPC}}$ . . . 34

C Chernoff Bound and Proof of Properties on Virtual Servers **35**

D The Protocol Description of Π_{share} **36**

E The Functionality $\mathcal{F}_{\mathsf{GLOSBaseROT}}$ **37**

F The Detailed Garbling Protocol for Virtual Servers **37**
- F.1 Subprotocols for the Garbling . . . 37
- F.2 Garbling Protocol Description . . . 38

G Description and Security Proof of $\mathsf{Sim}_{\mathsf{GLOS}}$ **40**
G.1 The Ideal World . . . 41
G.2 The Real World . . . 41
G.3 Description of $\mathsf{Sim}_{\mathsf{GLOS}}$ . . . 43
G.4 Security Proof . . . 48

H Security Proof for the Verifiable P2P Channel **58**

I Detailed Protocol Description of Π_{CompVer} **61**

J The Security Proof for the Main Protocol **62**

K The Cost Analysis for the Main Protocol **74**

1 Introduction

Secure multiparty computation (MPC) [Yao82, GMW87] allows n parties to jointly compute a function over their private inputs. The protocol ensures that an adversary corrupting up to t parties cannot learn any further information about honest parties' inputs beyond corrupted parties' outputs. There are various settings for the security guarantees of MPC protocols. Among all the security guarantees, *guaranteed output delivery* (GOD) is the strongest one. It ensures that, no matter how corrupted parties deviate from the protocol description, honest parties will eventually obtain their correct outputs. As a result, MPC protocols with GOD are much more difficult to build than those with weaker security guarantees, such as *security with abort*. In particular, GOD is only possible when a majority of the parties are honest [Cle86]. On the other hand, with an honest majority, GOD can even be achieved with *information-theoretic* security assuming a broadcast channel [RB89, Bea89].

In this work, we study MPC protocols with GOD in the honest majority setting, where $n = 2t + 1$, and aim to achieve constant round complexity that is independent of the circuit structure and the number of parties. It remains a big open question whether constant-round MPC with information-theoretic security and a polynomial amount of communication[1] is possible even for semi-honest security. To bypass this open question, we consider the *Minicrypt* world [Imp95] and assume the existence of a random oracle, which can be viewed as an ideal abstraction of one-way functions. Although many advances have been made over the years in enhancing the efficiency of MPC protocols with GOD in Minicrypt, existing solutions either require a non-constant number of rounds or incur a large communication overhead, especially in the standard honest majority setting with $n = 2t + 1$.

Communication-Efficient but Non-Constant-Round MPC with GOD. Since the early feasibility result of [RB89, Bea89], a line of works [CDD+99, BTH06, BFO12, GSZ20] improve the communication complexity of information-theoretic honest majority MPC with GOD regardless of round complexity. These works use the techniques of party elimination [HMP00] or dispute control [BTH06], which incur at least an additional $\Omega(n)$ term on the round complexity of a semi-honest MPC. The State-of-the-Art result [GSZ20] achieves communication of $O(|C|n)$ elements over P2P channels and $O(\mathsf{poly}(n))$ elements over broadcast channels, while the round complexity becomes $O(D + n^2)$. If non-black-box use of a pseudorandom function (PRF) is allowed, $O(|C| \cdot \mathsf{poly}(\log n, \kappa))$ communication complexity can be achieved within $\mathsf{poly}(n, \kappa)$ rounds [DIK+08] when there is a gap to the optimal resilience, i.e., $t = (1/2 - \epsilon)n$ for any constant $\epsilon > 0$.

There are also solutions that achieve $O(D)$-round honest majority MPC with GOD [CP17, ABDE24] in the information-theoretic setting. The best achieved communication complexity is $O(|C|n^3)$ via both P2P and broadcast channels [ABDE24], which is far less efficient than the solution with $\Omega(D + n)$ round complexity. It remains open whether linear communication can be achieved for GOD without a round complexity relying on n, even in Minicrypt. Moreover, these works all require a large communication cost over broadcast channels, where instantiating communication over broadcast channels is often harder than that over P2P channels. This further lowers the efficiency of their protocols.

Constant-Round MPC with GOD. There are also works achieving constant round complexity in honest majority MPC with GOD in Minicrypt. However, known solutions that make black-box use of the underlying cryptographic primitives either require exponential communication in the number of parties or only work for a sub-optimal corruption threshold (but are still not able to achieve linear communication):

- In the case $n = 2t + 1$, there are solutions [AKP22, AKP23] showing feasible results of round-optimal MPC with GOD in Minicrypt. However, the communication complexities of all the known results are exponential in the number of parties.
- In the case $n = 3t + 1$, there is an MPC with GOD that requires $O(|C|n^2\kappa)$ communication complexity and a constant round complexity [CGHZ16], which is constructed in the BMR framework [BMR90].

[1] Constant-round MPC with information-theoretic security is known from randomized encoding [ABT21] but requires an exponential communication cost to the depth of the circuit.

We note that a folklore solution is to compute a Yao's garbled circuit [Yao86, BMR90] using a generic $O(D)$-round MPC with GOD. However, this approach requires non-black-box use of one-way functions and further assumes a one-way function in $\mathbf{NC}^1$ (such that we can turn it into a constant-depth circuit using randomized encoding), and the communication cost is at least $O(|C|n^3\mathsf{poly}(\kappa))$ using the best known $O(D)$-round MPC in this setting [ABDE24].

As we can see, for MPC protocols with GOD in Minicrypt, there is a huge gap in the communication complexity between those with and without constant round complexity. This leads to the following question.

"Can we construct an honest majority MPC protocol in Minicrypt with guaranteed output delivery that achieves both constant round complexity and linear communication complexity in the number of parties?"

As we discussed above, this question *remains open* even if allowing $O(D)$ round complexity, where the best-known result [ABDE24] requires $O(|C|n^3)$ elements of communication over both P2P channels and broadcast channels, or relaxing to sub-optimal corruption threshold, where the best-known result [CGHZ16] requires $O(|C|n^2\kappa)$ bits of communication when $n = 3t + 1$.

1.1 Our Contribution

In this work, we give a positive answer to this question. Formally, we prove the following theorem:

Theorem 1 *Assuming broadcast channels and a random oracle, there is a computationally secure constant-round MPC protocol with GOD against a fully malicious adversary controlling up to $t = (n-1)/2$ parties with communication complexity $O(|C|n\kappa + Dn^3\kappa^3 + W_I \cdot \mathsf{poly}(n,\kappa))$ bits via P2P channels and $\mathsf{poly}(n,\kappa)$ bits via broadcast channels, where $|C|$ is the circuit size, D is the circuit depth, W_I is the number of input wires of C, and κ is the computational security parameter.*

To achieve this result, we give a compiler that compiles a recent multiparty garbling protocol [GLOS25] that only achieves security-with-abort to achieve guaranteed output delivery with linear communication. Our compiler enhances the underlying garbling protocol with identifiable abort, a prerequisite for the party elimination technique [HMP00] needed to achieve GOD. A core component in our construction is a verifiable P2P channel, which allows parties to either deliver the messages with guarantee or identify a malicious sender during interaction. The idea of the verifiable P2P channel comes from [IOZ14], while ours maintains a $\mathsf{poly}(n,\kappa)$ broadcast cost that is independent of the message size. To overcome the potential $O(n)$ round complexity from the party elimination technique, our verification phase is carefully designed to be able to identify a large set of malicious parties simultaneously. Furthermore, if the number of identified malicious parties is small, our protocol still guarantees the success of evaluation of the garbled circuit by using an error-correcting secret sharing scheme. Finally, we devise an efficient protocol for reconstructing the garbled circuit without incurring a quadratic communication overhead. We refer the readers to Section 2 for a detailed overview of our techniques.

Our result is a theoretical result targeting *asymptotic* communication efficiency *per gate* and constant round complexity. The concrete number of rounds and communication cost are still far from practice, and we leave the improvement of concrete efficiency to future work.

1.2 Other Related Works

There are also several related works in other settings, which we discuss below.

GOD MPC with 1/3 Corruption. In the setting of $n = 3t + 1$, GOD MPC can be constructed much easier than in the honest majority setting. In particular, perfect security with GOD can be achieved in this setting [BGW88, CCD88]. A few works [HMP00, BTH08, GLS19] then target at improving the communication complexity of MPC protocols in this setting, and finally achieve $O(|C|n)$ communication[2]. However,

[2]We omit the sublinear terms to $|C|$ in the communication complexity of the mentioned works in this section.

the achieved round complexity is $O(D+n)$, where D is the circuit depth. Later, [AAY21] achieves $O(|C|n^3)$ communication within $O(D)$ rounds, and the communication complexity is further optimized to $O(|C|n)$ by [AAPP23]. As we have mentioned, constant round complexity can be achieved in the computational setting assuming one-way functions when $n = 3t + 1$ [CGHZ16] in the BMR template [BMR90], requiring $\Omega(|C|n^2\kappa)$ communication complexity, where κ is the security parameter.

Efficient Honest Majority MPC without GOD. If we consider MPC protocols with weaker security guarantees, there exist honest majority protocols that are both round-efficient and communication-efficient. In particular, $O(|C|n)$ communication complexity and $O(D)$ round complexity can be achieved with either semi-honest security [DN07] or malicious security with abort [GIP+14] unconditionally. In the computational setting, constant round complexity can be achieved [BMR90] when one-way functions exist. In this setting, $O(|C|n\kappa)$ communication complexity can be achieved [LS25] for malicious security with abort. Further assuming random oracles, the best known result on constant-round MPC achieves $O(|C|\kappa)$ communication [GLOS25], which also achieves security with abort.

Efficient MPC from Stronger Assumptions. Under Decisional Diffie-Hellman (DDH) and Learning Parity with Noise (LPN) assumptions, linear-communication constant-round MPC can be achieved in the dishonest majority setting $t = n - 1$ [GLM+24]. When $t = (1/2 - \epsilon)n$, $O(|C|\kappa)$ communication complexity is achieved by [BGH+23][3] from the LPN assumption.

Round-Optimal Honest Majority MPC with GOD. There are also a line of works [AJL+12, GLS15, ACGJ18, BJMS20, AKP22, AKP23] that target optimizing the round complexity in honest majority MPC based on various assumptions. However, these works are all theoretical feasibility results focusing on the round complexity, without consideration of the communication complexity. Among them, the works in Minicrypt [AKP22, AKP23] both require an exponential communication cost in n.

2 Technical Overview

In this section, we give a high-level overview of our main techniques. Recall that our goal is to construct a constant-round MPC protocol with linear communication in the standard honest majority setting where $n = 2t + 1$. We focus on achieving guaranteed output delivery (GOD) against a fully malicious adversary. In particular, we consider communication over BC channels to be more expensive than over P2P channels, so we aim to maintain a $\mathsf{poly}(n, \kappa)$ broadcast message size.

2.1 Background: Constant-Size Multiparty Garbled Circuit

To build a constant-round secure multiparty computation protocol, the generic solution in the literature is to let all parties jointly generate a garbled circuit, which we refer to as a multiparty garbling protocol. Then each party acts as the evaluator and locally computes the function output. When the goal is to achieve malicious security with abort, it is sufficient to let a single party act as the evaluator, who computes the function output and then sends back the result to all parties, thus saving the communication of distributing the garbled circuit to all parties. This trick has been used in many works that focus on communication efficiency, such as [WRK17, YWZ20, BGH+23, GLM+24, HKN+25, GLOS25, LS25].

Unfortunately, for guaranteed output delivery, a single evaluator is not enough, as a corrupted evaluator may never send the result back. To achieve linear communication complexity, we need to construct a multiparty garbling protocol where the garbled circuit size is independent of the number of parties (so that after distributing the garbled circuit to all parties, the overall communication remains linear). Prior to [GLOS25], known solutions in Minicrypt (with black-box use of cryptographic primitives) all have garbled

[3]The work [BGH+23] achieves $O(|C|\kappa)$ communication complexity without a term related to the circuit depth D, while the previously mentioned work [GLOS25] has such a term.

circuit size $\Omega(Cn\kappa)$, even with sub-optimal corruption threshold. Therefore, just reconstructing such a garbled circuit to all parties already costs quadratic communication complexity, not even counting the cost of securely computing such a garbled circuit.

Fortunately, the recent multiparty garbling protocol [GLOS25] achieves malicious security with abort in the honest majority setting with $O(|C|\kappa + D \cdot \mathsf{poly}(n, \kappa))$ bits of both communication and garbled circuit size in the random oracle model. Our goal is to compile the protocol in [GLOS25] to achieve GOD.

The Multiparty Garbling Protocol in [GLOS25]. At a very high level, the garbled circuit is constructed via the round collapsing technique by garbling each party's next-message functions of a secret-sharing-based non-constant-round underlying MPC protocol Π_0 that meets certain requirements. Specifically, Π_0 is secure when up to $1/4$ of the parties are corrupted, and it only contains public reconstructions of packed secret sharings in the online phase. To boost the corruption threshold from $1/4$ to $1/2$, the authors in [GLOS25] use the party virtualization technique [Bra87, DIK+08]:

- Each virtual party in Π_0 is emulated by a random committee of c parties, where c is a constant.
- In each committee, all parties run a dishonest majority inner protocol to garble this virtual party's next-message functions in Π_0.

In the following, we use N to denote the number of virtual parties (or committees).

Our basic idea of achieving GOD in this work relies heavily on the structure of the multiparty garbling protocol in [GLOS25], but not the detailed construction. To better present our idea in this work, we only review the structure of their protocol. For a more detailed review of [GLOS25], we refer the readers to Section A. The multiparty garbling protocol in [GLOS25] contains a preprocessing phase, a garbling phase, and a verification phase:

Preprocessing Phase. In the beginning, parties prepare preprocessing data and input data, which includes (1) random linear secret sharings among all virtual parties, and (2) random OT correlations between each pair of parties.

Garbling Phase. Then, for each virtual party V_i, parties in the corresponding committee run a dishonest majority MPC protocol to garble the next-message functions of V_i in Π_0. In particular, the view of V_i is additively shared among the parties in this committee. The inner protocol is instantiated by a semi-honest dishonest majority multiparty garbling protocol based on the BMR template [BMR90, DI05] with the freeXOR optimization [KS08]. Parties in each committee may run such a dishonest majority MPC protocol using the prepared random OT correlations in the preprocessing phase in Minicrypt.

Verification Phase. To achieve malicious security, parties verify the validity of the preprocessing data and each virtual party's garbled circuits. This is done following the MPC-in-the-Head framework [IKOS07, IPS08]. To be more concrete, after the garbling phase, a constant fraction of the virtual parties' computation will be opened and verified by all parties. This ensures that most of the virtual parties' garbled circuits are correctly computed.

At the end of the protocol, each committee reconstructs the garbled circuits to an evaluator. Then the evaluator may emulate the underlying protocol Π_0 round by round by evaluating each virtual party's garbled circuits for his next-message functions. To be more concrete,

- In the beginning, the evaluator will in addition receive the wire labels of all virtual parties' first-round garbled circuits. This allows the evaluator to emulate all virtual parties' first-round computation.
- In the i-th round, the evaluator will obtain a packed secret sharing $[\boldsymbol{s}_i]$ by evaluating all virtual parties' garbled circuits as well as a packed secret sharing of the wire labels associated with $\boldsymbol{s}_i$. Then the evaluator can reconstruct $\boldsymbol{s}_i$ and their labels, which allows him to evaluate all virtual parties' $(i+1)$-th round garbled circuits.
- Finally, the evaluator computes the function output from all virtual parties' last-round garbled circuits.

2.2 Augmenting the Verification with Identifiable Abort

Now we return to our purpose of achieving GOD in a constant number of rounds. In the literature, a natural idea for achieving guaranteed output delivery is to use the party elimination technique [HMP00]. At a high level, it requires parties to identify a corrupted party who deviates from the protocol description when the computation fails. Then this party is removed, and the whole computation restarts.

To apply the party elimination technique, we need to first augment the verification phase to achieve an identifiable abort. At first glance, this task appears simple since during the verification, for each selected virtual party, all parties will open the views of parties in the corresponding committee and check the computation. Then, if the computation is incorrect, at least one party deviates from the protocol description, and by checking their views, we may expect to identify this corrupted party. Unfortunately, this is not the case due to the following two issues.

- First, the protocol may fail due to incorrect preprocessing data. Recall that in the preprocessing phase, all parties will prepare random OT correlations for each pair of parties, which are used to instantiate the inner dishonest majority MPC. If honest parties start with incorrect random OT correlations or corrupted parties change their preprocessing data, even if all parties follow the protocol, the output is still incorrect.

- Second, due to the use of P2P channels, it is possible that two parties do not agree on some message sent from one party to the other. In this case, although we know one of these two parties is corrupted, it is not clear which one is corrupted. A straightforward idea is to let the sender broadcast a commitment of his message and send the opening of the commitment to the receiver. Later, the receiver can use the opening of the commitment as a proof to show that this is indeed the message from the sender. Unfortunately, this does not work if the sender does not send a correct opening or even sends nothing to the receiver.

Verifiable P2P Channels. To resolve the above two issues, we first adapt the idea of [IOZ14] to construct a protocol that strengthens the P2P channel. which we refer to as a verifiable P2P channel. Specifically, the protocol has the following properties:

- At the end of the protocol, either the receiver obtains the message from the sender, or all parties identify the corrupted sender.

- When needed, the sender or the receiver can reveal this message to all parties.

The idea behind is to let the sender first broadcast a commitment and send the opening to the receiver. If the receiver does not receive the correct opening of the commitment, he will send a complaint message. Then, the sender will send the same opening to each party, who serves as an intermediate to pass the opening to the receiver. At the same time, all parties vote to decide whether the sender is corrupted. If $t+1$ parties claim they do not receive the correct opening from the sender, the sender is identified as a corrupted party. Otherwise, the sender must send the correct opening to at least one honest party, who will eventually pass it to the receiver. When needed, both the sender and the receiver can open the commitment to prove the validity of the message. We point out that this solution blows up the communication complexity by an $O(n)$ multiplicative overhead. However, since the underlying multiparty garbling protocol in [GLOS25] achieves $O(|C|\kappa)$ communication, we can still achieve $O(|C|n\kappa)$ communication.

Notably, the verifiable P2P channel constructed in [IOZ14] based on information-theoretic signatures [SHZI02, SASM10, SS11] requires a linear communication over broadcast channels in the size of the message sent via the verifiable channel. In contrast, the broadcast cost of our construction is independent of the message size. This allows us to maintain a $\mathsf{poly}(n, \kappa)$ communication cost via broadcast channels throughout the protocol execution.

Generating Verifiable Preprocessing Data. With verifiable P2P channels, we handle the preprocessing data in two different ways. Intuitively, we want to generate a verifiable version for each value in the preprocessing data.

- For random linear secret sharings, the dealer will distribute the shares using verifiable P2P channels. The dealer then proves the validity of his sharings by broadcasting a random linear combination of all random linear secret sharings. Each party verifies whether his shares match the broadcast secret sharing. If not, this party accuses the dealer by revealing the received shares to all parties. Then either the corrupted dealer or a corrupted accuser will be identified. For each corrupted dealer that is identified, all the shares he generated are set to 0.

- For random OT correlations, we will use a constant-round MPC protocol with GOD to let all parties prepare a *verifiable secret sharing* (VSS) for each value in the correlation. In this way, parties can reconstruct the verifiable secret sharings if some virtual party's computation is incorrect due to incorrect OT correlations. The key point is that we only need a small number of random OT correlations for each pair of parties who can use an OT extension protocol to obtain a sufficient number of random OTs. In addition, the circuit for preparing verifiable secret sharings for OT correlations has a constant depth. Thus, it is sufficient to use the MPC protocol in [CP17][4] which achieves $O(|C|n^4)$ communication and requires $O(D)$ rounds.

2.3 Handling Corrupted Parties Identified in the Verification

With an identifiable abort in the verification, we can identify one or more corrupted parties when the verification fails. However, the problem is what we should do next. Indeed, if we directly apply the party elimination technique, we may have to rerun the whole protocol $O(n)$ times since a single corrupted party may lead to failure in the verification, in which case only this single corrupted party can be identified. This will not only require $O(n)$ rounds, but also result in another $O(n)$ multiplication overhead in the communication complexity.

Our hope is to identify $\Theta(n)$ corrupted parties each time so that we only need to rerun the protocol a constant number of times. This requires our construction to satisfy the following two properties:

- First, the verification phase needs to identify all corrupted parties that deviate from the garbling phase.

- Second, if the number of identified corrupted parties is smaller than $t/2 = \Theta(n)$, parties can still obtain the function output without rerunning the protocol.

Identifying All Corrupted Parties. To achieve the first property, we first set the number of virtual parties (or committees) to be $N = \Theta(n\kappa)$, where recall that n is the number of real parties, and κ is the security parameter. Then, when each committee is randomly sampled, with overwhelming probability, every party participates in $\Theta(N/n)$ committees. During the verification phase, we will check a random subset of $\Theta(N) = \Theta(n\kappa)$ parties. Now, if a corrupted party behaves maliciously in a large number of committees, e.g., say $1/4$ of all committees he participates in, then with overwhelming probability, at least one such committee will be checked, and we can identify this corrupted party. From the union bound, with overwhelming probability, we can identify all corrupted parties who behave maliciously in many committees.

Running Evaluation Phase with the Rest of Virtual Parties. For each identified corrupted party, we will eliminate all committees this party participates in. Now if we have identified more than $t/2$ corrupted parties, we will rerun the whole protocol with a smaller corruption threshold following the party elimination framework. Otherwise, we want the rest of the parties to continue the evaluation phase and obtain the output.

[4]Here we do not use the MPC from [ABDE24] since it evaluates the circuit in a *detectable*-secret-shared fashion instead of a verifiable-secret-shared fashion.

Recall that in the evaluation phase of [GLOS25], after evaluating all virtual parties' garbled circuits in each round, the evaluator needs to reconstruct packed secret sharings to obtain the wire labels for the next-round garbled circuits. We note that the reconstruction can be done as long as the evaluator receives a sufficient number of shares. Thus, even if a small number of virtual parties fail to provide their garbled circuits, the evaluator can still compute the function output.

Using the above feature of [GLOS25], by carefully setting the parameters, we ensure that an honest evaluator can still compute the function output if no more than $t/2$ corrupted parties are eliminated.

One issue we omitted above is that, a corrupted party will only be identified with overwhelming probability if he behaves maliciously in *a large number* of committees. This means that the garbled circuits of the remaining virtual parties are not guaranteed to be correct. Fortunately, we can bound the number of incorrect garbled circuits since for each remaining corrupted party, he must honestly behave in most of the committees. In our construction, we will use a packed secret sharing scheme with error correction so that even if a small number of remaining virtual parties' garbled circuits are incorrect, the evaluator can still reconstruct the packed secret sharings in each round, and thus obtain the function output eventually.

Expander vs Random Graph. In both [GLOS25] and our construction, we use random committees to emulate virtual parties. This can be viewed as selecting the committees via a random bipartite graph, where n vertices are included in the left vertex set L and N vertices are included in the right one R. For each vertex in R, c randomly chosen edges connecting it are added to the graph. Then, the i-th committee is the index set of vertices in L that are connected to the i-th vertex in R. To boost the corruption threshold for virtual parties, we utilize the property that a random bipartite graph is a good expander: For any subset of $n - t$ nodes in L (corresponding to honest parties), the number of their neighbors in R (corresponding to committees that contain at least one honest party) is sufficiently large.

Thus, party virtualization can be used in boosting the threshold by using an explicit expander graph for committee selection [DIK+08]. In particular, an explicit expander graph can keep the number of virtual parties within $O(n)$, whereas in [GLOS25], they require at least $O(n + \kappa)$ committees. However, in both of [GLOS25] and our work, we also need the number of virtual parties to be sufficiently large ($\Omega(\kappa)$ in [GLOS25] and $\Omega(n\kappa)$ in our work) due to the MPC-in-the-Head verification. Therefore, using a random graph will not incur an additional overhead in the number of virtual parties. Moreover, we also need the underlying graph to be balanced in the sense that after eliminating a small number of vertices in L (corresponding to corrupted parties identified in the verification phase) and their neighbors in R, the number of remaining nodes in R is still sufficiently large, which is not guaranteed by a general expander graph. In addition, random graphs provide more convenience in proving the above requirements (using the Chernoff bound), so we use random graphs to select committees in our construction.

2.4 Reconstruction of Garbled Circuits to Evaluators

The above modifications, unfortunately, are still not sufficient to achieve guaranteed output delivery with linear communication. The final difficulty we face is reconstructing the garbled circuits to each party.

Recall that for each virtual party V_i, parties in the corresponding committee hold additive sharings of V_i's garbled circuits. To allow each party P_j to derive the function output, we need to reconstruct the remaining virtual parties' garbled circuits to P_j. However, there is no guarantee that a malicious party will send his additive shares to P_j. In the worst case, P_j may only receive shares from honest parties. Consequently, P_j may only obtain garbled circuits of virtual parties fully emulated by honest parties, which are insufficient for him to derive the function output.

This issue may look similar to the difficulty we face when augmenting the verification with an identifiable abort. Thus, one direct attempt is to use verifiable P2P channels. In this way, either the receiver will receive the message from the sender, or the malicious sender is identified. As we have pointed out in Section 2.2, this approach incurs an $O(n)$ multiplicative overhead in the communication complexity. While this is acceptable during the garbling phase, we cannot afford this blowup for distributing the garbled circuits since we have to reconstruct the garbled circuits of total size $O(|C|\kappa)$ to each party, which would require

$O(|C|n^2\kappa)$ communication due to the linear blowup!

Our solution is to reuse the structure of the random committees. After the verification phase, for each committee, we first reconstruct the garbled circuits of the emulated virtual party to each party in the committee. Since the committee size is a constant, we can afford the linear blowup in the communication complexity due to the use of verifiable P2P channels. Now we ask each party to send the garbled circuits he received to all parties using the standard P2P channels. Then as long as one honest party participates in the emulation of some virtual party V_i, every party will receive the garbled circuits of V_i. In our construction, we will choose the parameters so that the evaluation phase will succeed with garbled circuits from virtual parties which are (1) not eliminated in the verification phase, and (2) emulated by a committee that contains at least one honest party.

Another View of Our Solution. Our idea can be interpreted as follows: (1) First, we let each committee reconstruct the shared garbled circuits to a constant number of parties, and (2) Second, each party distributes all the garbled circuits he received in Step 1 to all parties. Since honest parties will always distribute their garbled circuits, we only need to ensure that after step 1, honest parties have received garbled circuits from a large fraction of committees.

Consider a bipartite graph where n vertices are included in the left vertex set L corresponding to the parties, and N vertices are included in the right vertex set R corresponding to the committees. An edge connects a left vertex P_j and a right vertex V_i if parties in the committee V_i reconstruct the shared garbled circuits to P_j. From this perspective, the property we need again translates to requiring this bipartite graph to be a good expander. Our solution simply reuses the graph derived from the committee selection, which is a good expander.

2.5 Summary of Our Protocol

One issue we omitted is how to deal with parties' inputs. In our construction, we may need to rerun the protocol twice due to the party elimination technique. To achieve malicious security, we need to ensure that parties use the same input in each attempt. Our solution is to let each party commit his input at the beginning using a constant-round *verifiable secret sharing* (VSS) protocol [CP17]. Then, we modify the functionality of the multiparty garbling protocol by taking VSSs of parties' inputs, reconstructing the secrets, and then computing the function output. Note that this incurs an overhead that depends on the input size of the circuit but is independent of the circuit size.

Let's finally summarize our protocol as follows:

1. At the beginning, all the parties share their inputs using VSS. Then, our goal is to construct an MPC for a circuit C' containing the VSS reconstructions of the inputs and the computation of the original circuit C.

2. The parties generate the preprocessing data for the garbling. In particular:

   (a) The parties run a general MPC [CP17] to compute base ROTs that are shared using verifiable secret sharings for each pair of parties.

   (b) The parties distribute the input/preprocessing shares to all the virtual parties, where each message is sent using the verifiable P2P channel. After the sharings are distributed, the parties compute and broadcast a random linear combination of the sharings generated by each party P_i. P_i himself also broadcasts the linear combination result and resolves the conflicts. Then, the parties check the validity of the sharings and eliminate the corrupted parties who do not generate valid sharings. All the shares generated by the parties eliminated in this step are set to 0.

3. The parties compute the garbled circuits. More concretely, parties follow the garbling phase of the protocol in [GLOS25], where each message is sent via the verifiable P2P channel. Finally, the parties broadcast the commitments for the labels corresponding to the preprocessing/input sharings and the shares for the garbled circuits.

4. The parties check each party P_i's computation in the garbling process and his linear combinations of preprocessing/input sharings. The parties agree on a small subset of virtual parties and open their computation. If the local computation of some party P_i is found not honestly performed, the parties eliminate P_i. If enough ($t/2 = \Theta(n)$) corrupted parties are eliminated at this point, the parties rerun the protocol from Step 2 with the rest of the parties, where the corruption threshold is set to be $t/2$.

5. The parties collect the garbled circuit. Each committee member collects the additive sharings of the corresponding virtual party's garbled circuits and input labels. The messages are also sent via the verifiable P2P channel in this step. Similarly, if enough ($t/2 = \Theta(n)$) corrupted parties are eliminated at this point, the parties rerun the protocol from Step 2 with the rest of the parties and a smaller corruption threshold. After the collection, each party sends the collected additive shares of garbled circuits to all other parties. Each party then checks whether the received shares match their commitments broadcast in the verifiable P2P channel and accepts the correct shares. Then, the parties are able to reconstruct a sufficient number of garbled circuits for the evaluation.

6. Finally, the parties locally do the evaluation and obtain their outputs.

3 Preliminaries

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

Security Model. In this work, we use the client-server model for secure multi-party computation, where $\tilde{n}$ clients and n servers run the protocol jointly. Among them, only clients have inputs and receive outputs. We assume the existence of both P2P channels and broadcast channels. Each message is associated with a message index mid during the protocol execution. We define the security of multiparty computation in the universal composability (UC) framework [Can01]. 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, where the PPT environment cannot distinguish the output distributions with more than $2^{-\Omega(\kappa)}$ advantage. For more details, we refer the readers to Section B.1.

3.1 Secret Sharing

Linear Secret Sharing Schemes. We follow the definition of linear secret sharing schemes (LSSSs) in [GLOS25], where an (N, T, k, ℓ)-LSSS over $\mathbb{F}_q$ is a secret sharing scheme that shares a secret vector in $\mathbb{F}_q^k$ to N shares, where each share is in $\mathbb{F}_q^\ell$. It is guaranteed that the joint distribution of any T shares is independent of the secret, and the sharing/reconstruction algorithms are all $\mathbb{F}_q$-linear.

We also define the multiplicative and error correction properties of an LSSS. We need an LSSS Σ with 3-multiplicative reconstruction, i.e., there exist LSSSs $\Sigma^{(2)}, \Sigma^{(3)}$ such that the share-wise tensor product of two Σ-sharings for secrets $\boldsymbol{s}^{(1)}, \boldsymbol{s}^{(2)}$ is a $\Sigma^{(2)}$-sharing for $\boldsymbol{s}^{(1)} * \boldsymbol{s}^{(2)}$, and similarly the share-wise tensor product of a Σ-sharing and a $\Sigma^{(2)}$-sharing is a $\Sigma^{(3)}$-sharing for the bit-wise product of their secrets. We also define that Σ has 3-multiplicative e-error-correction if any $N - 2e$ shares of an $\Sigma^{(3)}$-sharing uniquely determine its secret, such that the secret can be reconstructed even when $\leq e$ shares are incorrect.

In this work, we need to handle a set S of shares of an LSSS Σ, and these shares also need to form an LSSS. We call this kind of sharing a subsharing of Σ with respect to S. We give the formal definitions in Section B.2. Below is a lemma about the LSSS we use in our protocol construction.

Lemma 1 ([CC06]) *There exist constants ℓ, c such that for any integer n, $N > n$, $k \leq cn$ and $T \leq (n - 2e)/4$, there exists an (N, T, k, ℓ)-LSSS Σ such that any set S with $|S| = n, S \subset \{1, \ldots, N\}$, there exists a subsharing $\tilde{\Sigma}$ of Σ with respect to S with 3-multiplicative e-error-correction over $\mathbb{F}_2$.*

We use $[\cdot]$ to denote a Σ-sharing for the LSSS Σ from Lemma 1 and use $[\cdot]^{(2)}$ and $[\cdot]^{(3)}$ to denote the associated $\Sigma^{(2)}$ and $\Sigma^{(3)}$-sharings. Additionally, a vector of m Σ-sharings $([\boldsymbol{s}_1], \ldots, [\boldsymbol{s}_m])$ (resp. $\Sigma^{(2)}$-sharings $([\boldsymbol{s}_1]^{(2)}, \ldots, [\boldsymbol{s}_m]^{(2)})$, $\Sigma^{(3)}$-sharings $([\boldsymbol{s}_1]^{(3)}, \ldots, [\boldsymbol{s}_m]^{(3)})$) can naturally be regarded as an LSSS $\Sigma_{\times m}$ (resp. $\Sigma^{(2)}_{\times m}, \Sigma^{(3)}_{\times m}$) over $\mathbb{F}_{2^m}$. $\Sigma_{\times m}$ (resp. $\Sigma^{(2)}_{\times m}, \Sigma^{(3)}_{\times m}$) is called an m-fold interleaved secret sharing scheme of Σ (resp. $\Sigma^{(2)}, \Sigma^{(3)}$) [CCXY18], we denote it by $[\cdot]_{\times m}$ (resp. $[\cdot]^{(2)}_{\times m}, [\cdot]^{(3)}_{\times m}$).

Verifiable Secret Sharings. Verifiable Secret Sharing (VSS) [BGW88, RB89] is a basic protocol that enables a dealer to generate a sharing for a secret s to n parties such that any t parties' views are independent of the secret. Moreover, there is a reconstruction algorithm to reconstruct the secret from all the parties' views, even if t parties do not send their view correctly. We use $[\![s]\!]$ to denote a VSS for a secret s. A VSS can be modeled as a functionality $\mathcal{F}_{\mathsf{VSS}}$ that receives a secret bit s from the dealer and a request VSS from all the honest parties. Then $\mathcal{F}_{\mathsf{VSS}}$ generates a random VSS $[\![s]\!]$ for s and distributes it to all the servers. In particular, the adversary can choose the whole VSS if the dealer is corrupted, and it can choose corrupted servers' shares if the dealer is honest. For the formal definition of a VSS, we refer the readers to Section B.3. We provide the functionality $\mathcal{F}_{\mathsf{VSS}}$ as follows in Figure 1.

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

The functionality is run among a dealer D and n parties $P_1, \ldots, P_n$. Upon receiving VSS from all the honest parties and $s \in \{0, 1\}$ from the dealer D:

1. If D is corrupted, Sim receives the randomness $r \in \mathcal{R}$ for $\mathsf{VSS.Sh}$ from Sim. If D is honest, Sim receives the corrupted parties' shares of $[\![s]\!]$ from Sim and samples $r \in \mathcal{R}$ based on these shares.
2. The trusted party runs $\mathsf{VSS.Sh}(s; r)$ to obtain $(c_1, \ldots, c_n)$.
3. For each $i = 1, \ldots, n$, the trusted party sends c_i to P_i.

Figure 1: Functionality for a verifiable secret sharing.

In [RB89, CP17], the authors have shown that VSS can be instantiated in constant rounds within communication of $\mathsf{poly}(n, \kappa)$ bits under the condition $n = 2t + 1$.

Lemma 2 ([RB89, CP17]). *When $n = 2t + 1$, there exists a VSS and a protocol Π_{VSS} that t-securely realizes $\mathcal{F}_{\mathsf{VSS}}$ corresponding to the VSS with statistical security with communication of $\mathsf{poly}(n, \kappa)$ bits via both P2P channels and the broadcast channel in constant rounds.*

3.2 Basic Subprotocols

General MPC. We need the functionality $\mathcal{F}_{\mathsf{MPC}}$ of a general (non-constant-round) MPC protocol that takes as inputs a circuit C and its input wire values, and the functionality outputs VSSs of the circuit outputs. The functionality can be realized with communication of $O(|C|n^4)$ bits via both P2P and broadcast channels in $O(\mathsf{depth}(C))$ rounds [CP17]. We give the description of $\mathcal{F}_{\mathsf{MPC}}$ in Section B.4.

Random Coin. We also need the standard functionality $\mathcal{F}_{\mathsf{Coin}}$ which receives a request $\mathsf{RandCoin}$ from all honest parties and generates a public common coin in $\mathbb{F}_{2^\kappa}$. $\mathcal{F}_{\mathsf{Coin}}$ is provided in Figure 2.

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

On receiving $\mathsf{RandCoin}$ from all the honest parties, the trusted party samples $s \in \mathbb{F}_{2^\kappa}$ randomly and sends it to all the parties.

Figure 2: Functionality for generating a common coin.

$\mathcal{F}_{\mathsf{Coin}}$ can be computed by letting each party generate a random VSS and then reconstructing the sum of them. We state the lemma below.

Lemma 3 *$\mathcal{F}_{\mathsf{Coin}}$ can be t-securely realized with perfect security against a fully malicious adversary with communication of $\mathsf{poly}(n, \kappa)$ bits via both P2P channels and the broadcast channel in a constant number of rounds.*

4 The Structure of the Underlying Multiparty Garbling Protocol

In this section, we provide an abstract structure of the multiparty garbling protocol in [GLOS25] and give the properties it has. The structure and the properties will later be used in our compiler for compiling the protocol of [GLOS25] to our MPC protocol with guaranteed output delivery.

We make the following slight modifications to the original protocol of [GLOS25]:

- We will use different parameters from those in [GLOS25] to guarantee a stronger security, and we will give the parameter choices in this section.
- The chosen parameters enable an evaluator to obtain his circuit output from only a part of the resulting garbled circuit, and we also modify the evaluation process such that the garbled circuit does not need to be completely collected.

The protocol of [GLOS25] is constructed in the client-server model, where only clients have inputs and receive outputs. The only assumption used in the protocol of [GLOS25] is a random oracle (RO). In our main protocol, the (modified) protocol Π_{GLOS} of [GLOS25] may run twice with different sets of servers. For the first execution, all the servers in our main protocol will participate in Π_{GLOS}. That is to say, the number of servers is n and the corruption threshold is $t = (n-1)/2$. For the second execution, it starts only when over $t/2$ corrupted servers are eliminated. There remain $n' < n - t/2$ servers, where $t' = n' - t - 1 < n'/3$ servers among them may be corrupted.

We then begin to introduce the structure of Π_{GLOS}. At a high level, the MPC protocol Π_{GLOS} of [GLOS25] contains a preprocessing phase, a garbling phase, a verification process, and an evaluation phase, which achieves malicious security with abort. We only need the properties of Π_{GLOS} introduced in this section, and our compiler for achieving GOD does not rely on any other details about Π_{GLOS}. For the detailed description of Π_{GLOS}, we refer the readers to [GLOS25].

4.1 The Preprocessing Phase

We first introduce the structure of Π_{GLOS} from the preprocessing phase. In the preprocessing phase, the parties first determine the random committees, where the members of a committee will emulate a virtual server in the garbling phase. Then, the parties (including both clients and servers) distribute several linear secret sharings, where each virtual server's share is additively distributed among the committee members emulating the virtual server. The parties also prepare base random OTs (ROTs) to instantiate the dishonest majority MPC runs among each committee.

Determining the Virtual Servers. Our committee determination follows [GLOS25] to choose random committees of $c = \Theta(1)$ servers. This can be instantiated using a random public coin. We call a virtual server corrupted if it is emulated by a committee that is fully controlled by corrupted servers. The data of each virtual server is additively shared among its committee members, so the adversary only has full access to the corrupted virtual servers' data. The other virtual servers are called honest ones. Note that the computation of an honest virtual server can also be incorrect due to the corrupted parties in the committee. In [GLOS25], the number of random committees is chosen to be $N = \Theta(n + \kappa)$ to make sure that the proportion of corrupted virtual servers is small enough.

In this work, we additionally require enough virtual servers that perform the garbling honestly. We also need to identify the corrupted servers that deviate from too many inner protocols in which they participate. The corrupted servers are identified by opening a small fraction of virtual servers' views. To this end, it's necessary to ensure that each server emulates $\Omega(\kappa)$ virtual servers. Otherwise, even if a server emulating $o(\kappa)$

virtual servers does not honestly perform in all these inner protocols, the probability that he is identified (i.e., the probability that the view of a virtual server he emulate is required to be opened) is still $2^{-o(\kappa)}$. This is not enough.

Therefore, we increase the number of virtual servers to $N = \Theta(n\kappa)$ (and $N' = \Theta(n'\kappa)$) so that each server emulates $\Theta(\kappa)$ virtual servers on average. In particular, we take $c = 6$ and $N = 240n\kappa$[5] to ensure the following in the first execution of Π_{GLOS}:

- With an overwhelming probability, the number of corrupted virtual servers is no more than $N/60 = 4n\kappa$.
- For any set of $< t/2$ corrupted servers, with an overwhelming probability, the number of honest virtual servers completely emulated by the remaining $> n - t/2$ servers is at least $N/6 = 40n\kappa$.
- For any set of 3κ virtual servers, with an overwhelming probability, a randomly chosen set of $N/240 = n\kappa$ virtual servers includes a virtual server in this set.

Similarly, for the second execution, we also take $c = 6$ and set $N' = 240n'\kappa$. Then, it is guaranteed that:

- With an overwhelming probability, the number of corrupted virtual servers is no more than $N'/120 = 2n'\kappa$.
- With an overwhelming probability, the number of virtual servers completely emulated by honest servers is at least $N'/12 = 20n'\kappa$.
- For any set of 3κ virtual servers, with an overwhelming probability, a randomly chosen set of $N'/240 = n'\kappa$ virtual servers includes a virtual server in this set.

For each execution of Π_{GLOS}, the first property implies that the corrupted virtual servers only occupy a small fraction of all the virtual servers. The last property states that, if we open $1/240$ of the virtual servers' views, then each corrupted server can only maliciously behave in no more than 3κ virtual servers without being caught with a high probability. These properties upper-bound the number of virtual servers whose view can be accessed by the adversary and the number of virtual servers whose computation is incorrect. Together with the second property that lower-bounds the number of virtual servers whose computations are correctly computed, the protocol is guaranteed to proceed in the first execution when $< t/2$ corrupted servers are eliminated, and in the second execution.

These properties are proven by Chernoff bound. We provide more details in proving these properties in Section C.

Generating the Sharings. After the virtual servers are determined, the parties will run a sharing protocol Π_{Share}. Let Σ be the (N, T, k, ℓ)-LSSS given by Lemma 1 for the first execution of Π_{GLOS}, where $T = N/30$. For the second execution of Π_{GLOS}, Σ is set to be the (N', T', k', ℓ')-LSSS given by Lemma 1 with $T' = N'/60$.

In Π_{Share}, the parties only need to generate four kinds of sharing, including $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharings and random $\Sigma^{(2)}$-sharings with all-zero secrets. Then, the parties distribute these sharings to the virtual servers, where each virtual server's shares are additively shared among the servers emulating it.

Regardless of the verification and evaluation processes (which will be discussed in Section 4.3), the original protocol of [GLOS25] only requires Σ to have 3-multiplicative 0-error-correction, and the number of corrupted virtual servers should be at most T. Since our chosen parameters satisfy the required properties, the protocol remains secure after the modifications to the parameters. For completeness, we present the sharing protocol in Section D.

[5]Note that we aim to achieve $2^{-\Theta(\kappa)}$ error rate, and we do not focus on concrete efficiency in this work, so the constant in front of $n\kappa$ does not matter for the final result. Here, we choose $N = 240n\kappa$ only for convenience. Similar for N' below.

Preparing Base ROTs. The remaining preprocessing data required in [GLOS25] are the base random oblivious transfer correlations among each pair of servers $(S_{j,\alpha}, S_{j,\beta})$ in a committee, which is modeled by a functionality $\mathcal{F}_{\mathsf{GLOSBaseROT}}$ (which is only invoked once). We provide the formal functionality description of $\mathcal{F}_{\mathsf{GLOSBaseROT}}$ in Section E.

4.2 The Garbling Phase

We now briefly introduce the garbling phase. The garbling phase is run by the committees emulating the virtual servers. For each virtual server V_j, the servers $S_{j,1}, \ldots, S_{j,c}$ run a garbling protocol $\Pi_{\mathsf{Gb}}(V_j)$ with the following inputs:

1. All the preprocessing data for the virtual server prepared in the preprocessing phase, including the additively shared shares prepared in Π_{Share} and the base ROTs prepared between each pair of members received from $\mathcal{F}_{\mathsf{GLOSBaseROT}}$.
2. Each member $S_{j,\alpha}$'s uniformly random κ-bit seed $\mathsf{seed}_{j,\alpha}$ used for generating his randomness (via RO) in $\Pi_{\mathsf{Gb}}(V_j)$.

In particular, before running $\Pi_{\mathsf{Gb}}(V_j)$, each server commits to his inputs to $\Pi_{\mathsf{Gb}}(V_j)$ via the random oracle. Then, $S_{j,1}, \ldots, S_{j,c}$ jointly run $\Pi_{\mathsf{Gb}}(V_j)$. For completeness, we present the detailed garbling protocol $\Pi_{\mathsf{Gb}}(V_j)$ in Section F.

4.3 The Verification Phase

The verification phase of Π_{GLOS} is not directly used in our compiler, but we will need some properties on it. The verification phase is used to check that the corrupted servers perform honestly in $\Pi_{\mathsf{Gb}}(V_j)$ for most virtual servers V_j. Let the set of corrupted virtual servers be $\mathcal{T}_{\mathsf{cor}}$. In [GLOS25], the simulator $\mathsf{Sim}_{\mathsf{GLOS}}$ for the protocol Π_{GLOS} can provide the views of corrupted parties (including corrupted clients and servers) in the preprocessing phase and the garbling phase. Then, for a subset $\mathcal{T}_{\mathsf{ver}}$ containing at most $T - |\mathcal{T}_{\mathsf{cor}}|$ honest virtual servers, $\mathsf{Sim}_{\mathsf{GLOS}}$ can provide the views of honest servers during $\Pi_{\mathsf{Gb}}(V_j)$ for each $V_j \in \mathcal{T}_{\mathsf{ver}}$. This corresponds to the verification phase of [GLOS25] where all parties will open a random subset $\mathcal{T}_{\mathsf{ver}}$ of virtual parties' views to check the correctness of the computation. Finally, if there are fewer than $T - |\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}|$ honest virtual servers that do not follow the protocol (which is guaranteed with overwhelming probability if the verification passes), $\mathsf{Sim}_{\mathsf{GLOS}}$ can simulate the honest servers' outputs of $\Pi_{\mathsf{Gb}}(V_j)$ for all virtual servers V_j relying on one-time access to the main functionality $\mathcal{F}$.

Our compiler requires a stronger version of $\mathsf{Sim}_{\mathsf{GLOS}}$: We still require that $\mathsf{Sim}_{\mathsf{GLOS}}$ can provide the views of honest servers during $\Pi_{\mathsf{Gb}}(V_j)$ for each $V_j \in \mathcal{T}_{\mathsf{ver}}$ if $\mathcal{T}_{\mathsf{ver}}$ contains fewer than $T - |\mathcal{T}_{\mathsf{cor}}|$ honest virtual servers. Then, for a given subset $\mathcal{T}_{\mathsf{act}}$ of virtual servers, if there are fewer than $T - |\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}|$ honest virtual servers in $\mathcal{T}_{\mathsf{act}}$ that do not follow the protocol, $\mathsf{Sim}_{\mathsf{GLOS}}$ can simulate the honest servers' outputs of $\Pi_{\mathsf{Gb}}(V_j)$ for all virtual servers $V_j \in \mathcal{T}_{\mathsf{act}}$ relying on one-time access to $\mathcal{F}$.

Effectively, the original simulator in [GLOS25] works when $\mathcal{T}_{\mathsf{act}}$ is the set of all virtual servers, where the stronger version requires the same even when $\mathcal{T}_{\mathsf{act}}$ is a subset of all virtual servers. Note that this is a stronger requirement since honest virtual servers outside of $\mathcal{T}_{\mathsf{act}}$ may not follow the protocol, in which case it would not meet the condition required by the original simulator in [GLOS25].

Description of the Properties of $\mathsf{Sim}_{\mathsf{GLOS}}$. To be more concrete, we need the following three properties from $\mathsf{Sim}_{\mathsf{GLOS}}$ in our compiler.

First, $\mathsf{Sim}_{\mathsf{GLOS}}$ receives a subset $\mathcal{T}_{\mathsf{ver}}$ containing at most $T - |\mathcal{T}_{\mathsf{cor}}|$ (or $T' - |\mathcal{T}_{\mathsf{cor}}|$ for the second execution) honest virtual servers. $\mathsf{Sim}_{\mathsf{GLOS}}$ is able to generate the views of corrupted parties in the preprocessing phase and the garbling phase, including:

- All messages sent/received by a corrupted party during Π_{Share} and $\Pi_{\mathsf{Gb}}(V_j)$ for each virtual server V_j.

- The committed input (including the random tape) of each corrupted server $S_{j,\alpha}$ for the protocol $\Pi_{\mathsf{Gb}}(V_j)$. Our compiler does not use this information directly, but we need to use it to define the correct behaviors of an honest virtual server.

In addition, for each honest virtual server $V_j \in \mathcal{T}_{\mathsf{ver}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ is able to generate the views of honest servers involved in the emulation of V_j in $\Pi_{\mathsf{Gb}}(V_j)$.

Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ receives a subset $\mathcal{T}_{\mathsf{act}}$ of virtual servers and a set $\mathcal{T}_{\mathsf{hon}} \subseteq \mathcal{T}_{\mathsf{act}}$ of honest virtual servers such that:

1. $\mathcal{T}_{\mathsf{hon}} \cap (\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}}) = \emptyset$ and $|\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}| - |\mathcal{T}_{\mathsf{hon}}| \leq T$ for the first execution of Π_{GLOS} (replace T with T' for the second execution, same below).
2. For each $V_j \in \mathcal{T}_{\mathsf{hon}}$, servers honestly emulate V_j:
   - Each corrupted server among $S_{j,1}, \ldots, S_{j,c}$ runs $\Pi_{\mathsf{Gb}}(V_j)$ honestly from his committed input.
   - The shares generated in Π_{Share} for the virtual servers in $\mathcal{T}_{\mathsf{hon}}$ are all from valid sharings. More concretely:
     - For each sharing generated by an honest dealer in Π_{Share}, the shares sent to each corrupted server $S_{j,\alpha}$ (where $V_j \in \mathcal{T}_{\mathsf{hon}}$) match $S_{j,\alpha}$'s committed input to $\Pi_{\mathsf{Gb}}(V_j)$.
     - For each Σ-sharing generated by a corrupted dealer in Π_{Share}, there exists a valid Σ-sharing $[\boldsymbol{v}]$ such that for each $V_j \in \mathcal{T}_{\mathsf{act}}$, the shares of $\langle([\boldsymbol{v}])^{V_j}\rangle$ (where $([\boldsymbol{v}])^{V_j}$ is V_j's share of $[\boldsymbol{v}]$) for the honest servers among $S_{j,1}, \ldots, S_{j,c}$ match the messages sent by the corrupted dealer in Π_{Share}, and the shares for corrupted servers match their committed inputs to $\Pi_{\mathsf{Gb}}(V_j)$. Similar for the $\Sigma^{(2)}, \Sigma^{(3)}$-sharings and the $\Sigma^{(2)}$-sharings with all-zero secrets.
   - The input base ROTs (to $\Pi_{\mathsf{Gb}}(V_j)$) between each pair of servers $(S_{j,\alpha}, S_{j,\beta})$ emulating a virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ match what they received from $\mathcal{F}_{\mathsf{GLOSBaseROT}}$.

After receiving the above sets, $\mathsf{Sim}_{\mathsf{GLOS}}$ can generate the honest servers' outputs of $\Pi_{\mathsf{Gb}}(V_j)$ for each $V_j \in \mathcal{T}_{\mathsf{act}}$. The output by $\mathsf{Sim}_{\mathsf{GLOS}}$ is computationally indistinguishable from those in the real world.

As we have mentioned above, the original simulator in [GLOS25] works by randomly choosing $\mathcal{T}_{\mathsf{ver}}$ and taking $\mathcal{T}_{\mathsf{act}}$ to be the set of all virtual servers. If the verification passes, the simulator in [GLOS25] can extract out a subset $\mathcal{T}_{\mathsf{hon}}$ satisfying the above requirements with overwhelming probability. In our compiled protocol, we will run a mildly different version of the verification process. In the security proof of our protocol, we will build a simulator which can provide $\mathcal{T}_{\mathsf{ver}}, \mathcal{T}_{\mathsf{act}}$ and extract out the set $\mathcal{T}_{\mathsf{hon}}$ for $\mathsf{Sim}_{\mathsf{GLOS}}$ if the verification passes.

Existence of Stronger Simulator in [GLOS25]. We give a brief argument about the existence of the above stronger version of $\mathsf{Sim}_{\mathsf{GLOS}}$ for Π_{GLOS} in [GLOS25]. We observe that the original simulator of Π_{GLOS} works as follows.

1. The simulator first simulates the preprocessing phase and the garbling phase without relying on honest virtual servers' inputs.
2. For at most $T - |\mathcal{T}_{\mathsf{cor}}|$ honest virtual servers, the simulator is able to generate the entire views of involved honest servers in the emulation of these virtual servers' computation in the garbling phase. This is achieved by explicitly generating these honest servers' inputs and randomness and then generating the entire views of honest servers relying on corrupted servers' views and the ability to program the output of the random oracle. The key observation is that for each honest virtual server V_j, the honest servers' inputs to $\Pi_{\mathsf{Gb}}(V_j)$ only contain their shares from Π_{Share} and their correlated randomness from random OTs. We can generate honest servers' inputs for $T - |\mathcal{T}_{\mathsf{cor}}|$ honest virtual servers relying on the secrecy of the underlying Σ-sharing scheme.

   Using this property, the simulator is able to first generate honest servers' views for virtual servers in $\mathcal{T}_{\mathsf{ver}}$ and then generate honest servers' views (which include their outputs) for virtual servers in $\{V_1, \ldots, V_N\} \backslash (\mathcal{T}_{\mathsf{hon}} \cup \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}})$.

3. Finally, the simulator simulates the outputs of $\Pi_{\mathsf{Gb}}(V_j)$ for each other honest virtual server V_j (excluding the honest virtual servers whose complete views are simulated in Step 2) without generating the honest servers' inputs to it.

In [GLOS25], the verification phase ensures that if the verification passes, then no more than $T-|\mathcal{T}_{\mathsf{cor}}|$ honest virtual servers' computations are incorrect. Therefore, the simulator will simulate the entire inputs of these honest virtual servers to simulate their outputs, and only simulate the correct outputs for the rest of the honest virtual servers. On the other hand, if the verification fails, then the protocol would abort, and there is no need to simulate any honest virtual server's output (thus no need to generate honest virtual servers' inputs for those whose computations are incorrect).

In our construction, to achieve GOD, if only a limited number of corrupted parties are identified, the protocol will continue by excluding all virtual servers that involve any of the identified corrupted parties. The remaining virtual servers correspond to $\mathcal{T}_{\mathsf{act}}$, and we still need the simulator to produce the outputs of honest virtual servers in $\mathcal{T}_{\mathsf{act}}$ in this case. We note that the above-mentioned property of the simulator in [GLOS25] suffices. In $\mathcal{T}_{\mathsf{act}}$, only for honest virtual servers whose computations are incorrect, the simulator needs to simulate their inputs explicitly (to generate their outputs). For the rest of the honest virtual servers in $\mathcal{T}_{\mathsf{act}}$, it is sufficient to only simulate their correct outputs. Such a simulator can be induced from the simulator in [GLOS25]. More concretely, in Step 2, $\mathsf{Sim}_{\mathsf{GLOS}}$ first generates honest servers' views for virtual servers in $\mathcal{T}_{\mathsf{ver}}$ and then generates honest servers' views (which include their outputs) for virtual servers in $\mathcal{T}_{\mathsf{act}}\backslash(\mathcal{T}_{\mathsf{hon}}\cup\mathcal{T}_{\mathsf{ver}}\cup\mathcal{T}_{\mathsf{cor}})$. By the requirements of $\mathcal{T}_{\mathsf{ver}},\mathcal{T}_{\mathsf{act}},\mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ generates honest servers' views for at most $T-|\mathcal{T}_{\mathsf{cor}}|$ honest virtual servers. Then in Step 3, $\mathsf{Sim}_{\mathsf{GLOS}}$ is able to generate honest servers' outputs for each $V_j\in\mathcal{T}_{\mathsf{hon}}$.

However, a subtle issue is that in [GLOS25], servers execute a maliciously secure OT-extension protocol outside of $\Pi_{\mathsf{Gb}}(V_j)$ and only take random OTs to instantiate $\Pi_{\mathsf{Gb}}(V_j)$. As a result, honest servers' views for virtual servers in $\mathcal{T}_{\mathsf{ver}}$ simulated by the original simulator in [GLOS25] *do not* include their views in the OT-extension protocol. Our construction, on the other hand, requires opening the entire views of the OT-extension protocol for virtual servers in $\mathcal{T}_{\mathsf{ver}}$ (but not for virtual servers in $\mathcal{T}_{\mathsf{act}}\backslash(\mathcal{T}_{\mathsf{hon}}\cup\mathcal{T}_{\mathsf{ver}}\cup\mathcal{T}_{\mathsf{cor}})$) as well to identify corrupted servers.

We address this issue by providing $\mathcal{T}_{\mathsf{ver}}$ to $\mathsf{Sim}_{\mathsf{GLOS}}$ in the beginning so that for each $V_j\in\mathcal{T}_{\mathsf{ver}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ simply generates honest servers' inputs and follows the protocol in $\Pi_{\mathsf{Gb}}(V_j)$ to obtain honest servers' views. On the other hand, this requires the simulator of our construction to decide $\mathcal{T}_{\mathsf{ver}}$ before invoking $\mathsf{Sim}_{\mathsf{GLOS}}$. We show that this is not an issue. Intuitively, $\mathcal{T}_{\mathsf{ver}}$ is a random subset, which will be randomly sampled by the simulator in the ideal world. Thus, we simply let the simulator sample $\mathcal{T}_{\mathsf{ver}}$ in advance before invoking $\mathsf{Sim}_{\mathsf{GLOS}}$. We refer the readers to Section G for the formal description and security proof of $\mathsf{Sim}_{\mathsf{GLOS}}$.

4.4 The Evaluation Phase

The evaluation phase $\Pi_{\mathsf{Ev}}(P)$ of a client P takes the outputs of $\Pi_{\mathsf{Gb}}(V_j)$ for all virtual servers V_j in a set $\mathcal{E}_P\subseteq\mathcal{T}_{\mathsf{act}}$ which is locally generated by P. Let $e=|\mathcal{T}_{\mathsf{act}}\backslash(\mathcal{T}_{\mathsf{hon}}\cup\mathcal{T}_{\mathsf{ver}})|$. It is guaranteed that an honest client P can correctly evaluate his own output[6] through $\Pi_{\mathsf{Ev}}(P)$ with an overwhelming probability when the subsharing of Σ with respect to $\mathcal{E}_P$ has the property of 3-multiplicative e-error-correction.

This property is obtained from the fact that in the evaluation phase of [GLOS25], the evaluator emulates the underlying non-constant-round protocol Π_0 round by round by evaluating each virtual server's garbled circuits for his next-message functions. In particular, in each round, the evaluator only needs to reconstruct the secrets of several $\Sigma^{(2)},\Sigma^{(3)}$-sharings, where each virtual server's shares come from the evaluation of his garbled circuit. By the definition of $\mathcal{T}_{\mathsf{act}},\mathcal{T}_{\mathsf{hon}},\mathcal{T}_{\mathsf{ver}}$, at most $e=|\mathcal{T}_{\mathsf{act}}\backslash(\mathcal{T}_{\mathsf{hon}}\cup\mathcal{T}_{\mathsf{ver}})|$ virtual servers may generate incorrect garbled circuits. When the subsharing of Σ with respect to $\mathcal{E}_P$ has the property of 3-multiplicative e-error-correction, the evaluator can always reconstruct the correct secret for each of $\Sigma^{(2)},\Sigma^{(3)}$-sharing, thus obtain his correct function output.

[6] Here, "correctly" means that the output computed by P is the same as the output of P computed by the MPC functionality $\mathcal{F}$ after $\mathsf{Sim}_{\mathsf{GLOS}}$ sends the corrupted clients' inputs to $\mathcal{F}$.

5 Verifiable P2P Channels

In this section, we will construct the verifiable P2P channel, which is a key component of our compiler. Our aim is to let the channel provide commitments for the messages sent via it. The sender and the receiver of each message either agree on the commitment or one of them will be found corrupted and eliminated. We also allow the parties to request the sender or the receiver and let them open the message. Let mid_m denote the message index of a message m. We provide the following functionality for the P2P channel between two parties A and B, where A is the sender and B is the receiver.

Functionality $\mathcal{F}_{\mathsf{VP2P}}(A, B)$

Send a Message. Upon receiving $(\mathsf{send}_{A,B}, \mathsf{mid}_m)$ from all honest parties:

1. The trusted party then receives a message m from A.
2. If A is honest, the trusted party sends m to B and sends $(\mathsf{committed}, \mathsf{mid}_m)$ to all parties. Otherwise, the trusted party receives a request from Sim:
   - If the request is $(\mathsf{deliver}, \mathsf{mid}_m)$, then the trusted party sends m to B and sends $(\mathsf{committed}, \mathsf{mid}_m)$ to all parties (including clients or servers, same below).
   - Otherwise, the trusted party sends $(\mathsf{eliminate}, A)$ to all the parties.

Open a Message. Upon receiving $(\mathsf{open}_A, \mathsf{mid}_m)$ (resp. $(\mathsf{open}_B, \mathsf{mid}_m)$) from all honest parties, if m has been delivered to B, then:

1. If A (resp. B) is honest, the trusted party sends m to all the parties.
2. If A (resp. B) is corrupted, the trusted party receives a request from Sim:
   - If the request is $(\mathsf{open}, \mathsf{mid}_m, S)$, where $S \neq \emptyset$ is a set of honest parties, the trusted party sends m to all the parties in S and sends $(\mathsf{eliminate}, A)$ (resp. $(\mathsf{eliminate}, B)$) to all the other parties.
   - Otherwise, the trusted party sends $(\mathsf{eliminate}, A)$ (resp. $(\mathsf{eliminate}, B)$) to all the parties.

Figure 3: Functionality for the verifiable P2P channel between A and B.

Now we give the instantiation $\Pi_{\mathsf{VP2P}}(A, B)$ of the verifiable P2P channel.

Protocol $\Pi_{\mathsf{VP2P}}(A, B)$

Send a Message: To send a message m with message index mid_m from A to B:

1. A samples a κ-bit random string r, broadcasts $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ to all the servers, and sends m, r to B.
2. B checks whether m matches $\mathcal{O}(m\|r\|\mathsf{mid}_m)$. If not, B broadcasts $(\mathsf{complain}, \mathsf{mid}_m, B, A)$. Otherwise, B outputs m.
3. If $(\mathsf{complain}, \mathsf{mid}_m, B, A)$ is received:
   (a) A sends the message m with the randomness r corresponding to the message index mid_m to all the servers.
   (b) Each server checks whether m matches $\mathcal{O}(m\|r\|\mathsf{mid}_m)$. If not, the party broadcasts $(\mathsf{eliminate}, A)$.
   (c) If $(\mathsf{eliminate}, A)$ is received from $\geq t + 1$ servers, the servers output $(\mathsf{eliminate}, A)$. Otherwise, the servers deliver (m, r) to B. B checks whether the received (m, r) matches $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ and outputs m if they match. All the parties then output $(\mathsf{committed}, \mathsf{mid}_m)$.

   Otherwise, all the parties output $(\mathsf{committed}, \mathsf{mid}_m)$.

Open a Message: When A (resp. B) is requested to open a message m with message index mid_m:

1. A (resp. B) sends the message m with the randomness r corresponding to the message index mid_m to all the servers.
2. Each server checks whether m matches $\mathcal{O}(m\|r\|\mathsf{mid}_m)$. If not, the server broadcasts $(\mathsf{eliminate}, A)$ (resp. $(\mathsf{eliminate}, B)$). Otherwise, the server outputs m.

3. If $(\mathsf{eliminate}, A)$ (resp. $(\mathsf{eliminate}, B)$) is received from $\geq t+1$ servers, the servers output $(\mathsf{eliminate}, A)$ (resp. $(\mathsf{eliminate}, B)$).

Figure 4: The instantiation of the verifiable P2P channel between A and B.

Lemma 4 *$\Pi_{\mathsf{VP2P}}(A, B)$ t-securely realizes $\mathcal{F}_{\mathsf{VP2P}}(A, B)$ with computational security.*

To send or open a message m via $\Pi_{\mathsf{VP2P}}(A, B)$, the parties need to communicate $O(|m|n + \mathsf{poly}(n, \kappa))$ bits via P2P channels and $\mathsf{poly}(n, \kappa)$ bits via broadcast channels. We provide a proof of Lemma 4 in Section H.

6 Compiler for Achieving GOD

In this section, we give our compiler that compiles the protocol Π_{GLOS} of [GLOS25] to achieve guaranteed output delivery. In particular, we assume a random oracle $\mathcal{O}$ in our compiler, which is different from the random oracle used in Π_{GLOS}[7].

6.1 High-Level Structure

At a high level, our compiler uses the party elimination technique [HMP00]. We provide a compiled version of Π_{GLOS} such that it can achieve GOD in the case $n = 2t+1$ when fewer than $t/2$ corrupted servers are caught and eliminated. In addition, when more than $t/2$ corrupted servers are eliminated, the compiled version of Π_{GLOS} can also achieve GOD when it is run among the remaining n' parties. The high-level structure of the protocol is as follows:

1. First, the clients share their inputs to the servers via VSSs.
2. Then, the servers will serve as n clients that provide inputs to a circuit C' that takes the VSSs generated in the first step as its inputs. The $\tilde{n}$ real clients serve as another $\tilde{n}$ clients that only receive outputs from C'. The circuit C' first reconstructs the secrets of these VSSs and then evaluates C with the clients' inputs to C. In this way, each client's output from C is the same as his output from C'.
3. All the servers run the first execution of (the compiled version of) Π_{GLOS} to compute the circuit C' with their VSSs of the clients' inputs. If fewer than $t/2$ servers are eliminated, the clients obtain their outputs eventually. Otherwise, once more than $t/2$ servers are eliminated, the remaining servers run the second execution of (the compiled version of) Π_{GLOS}. Finally, the clients are still able to obtain their outputs.

In the next few subsections, we will introduce the compiled version of Π_{GLOS}.

6.2 Compiling the Preprocessing

We first show how to compile the preprocessing phase of Π_{GLOS}. Recall that the preprocessing phase contains the determination of virtual servers, the sharing phase Π_{Share}, and the invocation of $\mathcal{F}_{\mathsf{GLOSBaseROT}}$. We follow Π_{GLOS} to determine the virtual servers, where the parameter choices are given in Section 4.1.

Preparing Sharings. For Π_{Share}, all the messages are sent via $\mathcal{F}_{\mathsf{VP2P}}$. All the shares are committed in this way. In particular, the messages from a party P (either a client or a server) to a member $S_{j,\alpha}$ of V_j are sent together with a single message index via $\mathcal{F}_{\mathsf{VP2P}}(P, S_{j,\alpha})$, while the messages from P to $S_{j_1,\alpha_1} \neq S_{j_2,\alpha_2}$ are not sent together, no matter whether S_{j_1,α_1} and S_{j_2,α_2} are emulated by one real server or not.

Let $\kappa' = N + \kappa$. After Π_{Share} is executed in the above way, we let each party P generate a random $\Sigma_{\times\kappa'}$-sharing $[\boldsymbol{r}_1^P]_{\times\kappa'}$, a random $\Sigma^{(2)}_{\times\kappa'}$-sharing $[\boldsymbol{r}_2^P]^{(2)}_{\times\kappa'}$, a random $\Sigma^{(3)}_{\times\kappa'}$-sharing $[\boldsymbol{r}_3^P]^{(3)}_{\times\kappa'}$, and a random $\Sigma^{(2)}_{\times\kappa'}$-sharing $[\boldsymbol{o}^P]^{(2)}_{\times\kappa'}$ with an all-zero secret. Then, P distributes them to the virtual servers via $\mathcal{F}_{\mathsf{VP2P}}$. These

[7] This can be instantiated with a single random oracle easily by adding a prefix before each query made in Π_{GLOS}.

sharings are used for masking the sharings generated by P in Π_{Share} during the later verification of these sharings.

Preparing Base ROTs. For the step of preparing the base ROTs, we make a slight modification to the ROT functionality to make the ROT results committed. We let the functionality not only output the base ROT correlations but also output the verifiable secret sharings of them. The VSSs provide commitments to the base ROT outputs. The modified functionality $\mathcal{F}_{\mathsf{BaseROT}}$ is presented as follows.

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

Upon receiving ROT from all the honest servers, for each pair of members $(S_{j,\alpha}, S_{j,\beta})$ of each virtual server V_j, the trusted party does the following κ times:

1. The trusted party randomly samples two messages $r_0, r_1 \in \{0,1\}^\kappa$. and a bit $b \in \{0,1\}$. If $S_{j,\alpha}$ is corrupted, r_0, r_1 are chosen by Sim. If $S_{j,\beta}$ is corrupted, b, r_b are chosen by Sim.
2. The trusted party receives a set of corrupted parties' shares of $[\![b]\!], [\![r_0]\!], [\![r_1]\!], [\![r_b]\!]$ from Sim. Then, the trusted party samples the whole sharings $[\![b]\!], [\![r_0]\!], [\![r_1]\!], [\![r_b]\!]$ based on the secrets and the corrupted parties' shares.
3. The trusted party sends (r_0, r_1) to $S_{j,\alpha}$ and sends (b, r_b) to $S_{j,\beta}$. The trusted party also sends each party P_i's shares of $[\![b]\!], [\![r_0]\!], [\![r_1]\!], [\![r_b]\!]$ to P_i.

Figure 5: Functionality for base random oblivious transfers.

$\mathcal{F}_{\mathsf{BaseROT}}$ can be instantiated using a general VSS-based MPC with GOD [CP17]. More concretely, to prepare each ROT correlation $((r_0, r_1), (b, r_b))$ between server pair $(S_{j,\alpha}, S_{j,\beta})$, we only need to let $S_{j,\alpha}$ input two random strings as r_0, r_1 to $\mathcal{F}_{\mathsf{MPC}}$ and let $S_{j,\beta}$ input a random bit as b to $\mathcal{F}_{\mathsf{MPC}}$. The circuit for computing $\mathcal{F}_{\mathsf{BaseROT}}$ is of $\mathsf{poly}(n, \kappa)$ size and constant depth, so we get the following lemma.

Lemma 5 *$\mathcal{F}_{\mathsf{BaseROT}}$ can be t-securely realized with statistical security against a fully malicious adversary with communication of $\mathsf{poly}(n, \kappa)$ bits via both P2P channels and the broadcast channel in constant rounds.*

6.3 Verification of the Sharings

In our compilation, we verify the sharings before the garbling. Recall that the sharings generated in Π_{Share} only consist of four types of sharings, $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharings and random $\Sigma^{(2)}$-sharings with all-0 secrets. We let all servers check a random linear combination of each type of sharings generated by each party P, where the result is masked by the auxiliary sharing distributed by P. The validity of the resulting sharing is checked by all parties. When the resulting sharing is invalid, we let all the parties mark P "eliminated".

After that, one more thing the servers need to do is to cross-check the shares, which guarantees that the linear combinations of honest servers' shares are all correctly computed. We let each server also broadcast the linear combination of his shares received from the checked party P[8]. If a conflict is found, the two parties are required to open the shares sent via $\mathcal{F}_{\mathsf{VP2P}}$, and the party whose opening does not match the linear combination result is marked "eliminated". Finally, for each eliminated party P, the sharings generated by P are set to be all-zero. The protocol Π_{ShVer} for verifying the validity of the sharings is given in Figure 6.

Protocol Π_{ShVer}

For each party P, let all the Σ-sharings he generated in Π_{Share} be $[\boldsymbol{x}^P_{1,1}], \ldots, [\boldsymbol{x}^P_{1,k_1}]$, all the $\Sigma^{(2)}$-sharings he generated in Π_{Share} be $[\boldsymbol{x}^P_{2,1}]^{(2)}, \ldots, [\boldsymbol{x}^P_{2,k_2}]^{(2)}$, all the $\Sigma^{(3)}$-sharings he generated in Π_{Share} be $[\boldsymbol{x}^P_{3,1}]^{(3)}, \ldots, [\boldsymbol{x}^P_{3,k_3}]^{(3)}$, and all the $\Sigma^{(2)}$-sharings with all zero secrets he generated in Π_{Share}be $[\boldsymbol{o}^P_1]^{(2)}, \ldots, [\boldsymbol{o}^P_{k_4}]^{(2)}$. Let $\kappa' = N + \kappa$. The servers view the sharings to be checked as $\Sigma_{\times\kappa'}, \Sigma^{(2)}_{\times\kappa'}, \Sigma^{(3)}_{\times\kappa'}$-sharings (each $[\boldsymbol{x}^P_{1,i}]$ is viewed as $[\boldsymbol{x}^P_{1,i}]_{\times\kappa'}$, similar for $\Sigma^{(2)}_{\times\kappa'}, \Sigma^{(3)}_{\times\kappa'}$-sharings) by simply embedding each bit into $\mathbb{F}_{2^{\kappa'}}$.

[8] We can also let only P open the sharing and all parties verify their shares locally. The communication cost is similar, but requires an extra round to let the parties complain.

Then:

1. The servers invoke $\mathcal{F}_{\mathsf{Coin}}$ to get $s_{\mathsf{ver}} \in \mathbb{F}_{2^\kappa}$. Then the servers expand s_{ver} to a vector $(s_{1,1}, \ldots, s_{1,k_1}, s_{2,1}, \ldots, s_{2,k_2}, s_{3,1}, \ldots, s_{3,k_3}, s_{4,1}, \ldots, s_{4,k_4}) \in \mathbb{F}_{2^{\kappa'}}^{k_1+k_2+k_3+k_4}$ by input s_{ver} to a random oracle.
2. For each party P, P and the servers $S_{\alpha,1}, \ldots, S_{\alpha,c}$ of each virtual server V_α locally compute an additive sharing of V_α's shares of
$$[\tau_1^P]_{\kappa'} = \sum_{j=1}^{k_1} s_{1,j} \cdot [\boldsymbol{x}_{1,j}^P]_{\kappa'} + [\boldsymbol{r}_1^P]_{\kappa'},$$
$$[\tau_2^P]_{\kappa'}^{(2)} = \sum_{j=1}^{k_2} s_{2,j} \cdot [\boldsymbol{x}_{2,j}^P]_{\kappa'}^{(2)} + [\boldsymbol{r}_2^P]_{\kappa'}^{(2)},$$
$$[\tau_3^P]_{\kappa'}^{(3)} = \sum_{j=1}^{k_3} s_{3,j} \cdot [\boldsymbol{x}_{3,j}^P]_{\kappa'} + [\boldsymbol{r}_3^P]_{\kappa'}^{(3)},$$
and
$$[\tau_4^P]_{\kappa'}^{(2)} = \sum_{j=1}^{k_4} s_{4,j} \cdot [\boldsymbol{o}_j^P]_{\kappa'}^{(2)} + [\boldsymbol{o}^P]_{\kappa'}^{(2)}.$$
Then, they broadcast the results.
3. All the servers check whether the sharings $[\tau_1^P]_{\kappa'}, [\tau_2^P]_{\kappa'}^{(2)}, [\tau_3^P]_{\kappa'}^{(3)}, [\tau_4^P]_{\kappa'}^{(2)}$ from P's broadcast are valid $\Sigma, \Sigma^{(2)}, \Sigma^{(3)}$-sharings and a $\Sigma^{(2)}$-sharing with an all-zero secret respectively. If not, all the servers mark P "$\mathsf{eliminated}$".
4. If the broadcast by P does not match the broadcast by another party P', let the index of the message carrying all the shares sent from P to P' in Π_{Share} be mid. All the parties send $(\mathsf{open}_P, \mathsf{mid})$ to $\mathcal{F}_{\mathsf{VP2P}}(P, P')$. If $(\mathsf{eliminated}, P)$ is received by a party, the party broadcasts $(\mathsf{eliminate}, P)$. Each party that receives the opened shares verifies whether they match the broadcast from P. If not, the party broadcasts $(\mathsf{eliminate}, P)$. If $(\mathsf{eliminate}, P)$ is received from $t+1$ parties, the parties mark P "$\mathsf{eliminated}$". For each party P that is marked "$\mathsf{eliminated}$", all the servers regard that they receive all-zero shares from P.

Figure 6: The verification protocol for the sharings.

Note that, for the shares generated by a corrupted P to the corrupted parties, P may still not compute the linear combinations of these shares honestly. Thus, we should later verify the parties' local computations during Π_{ShVer} (together with the servers' local computations in the garbling phase).

6.4 Compiling the Garbling Phase

We then show how to compile the garbling phase. Before the garbling protocol $\Pi_{\mathsf{Gb}}(V_j)$ for each virtual server V_j is executed, each server $S_{j,\alpha}$ will additionally commit to his inputs for the garbling. Since the inputs to the garbling, i.e., the preprocessing data from Π_{Share} (that are sent via $\mathcal{F}_{\mathsf{VP2P}}$) and the base-ROTs from $\mathcal{F}_{\mathsf{BaseROT}}$ have all been committed, the parties only need to additionally commit to their random seeds. More concretely, after executing the preprocessing, each server $S_{j,\alpha}$ who emulates a virtual server V_j samples the random seed $\mathsf{seed}_{j,\alpha} \in \mathbb{F}_2^\kappa$ as his input seed to $\Pi_{\mathsf{Gb}}(V_j)$ and another random κ-bit string $s_{j,\alpha}$. Then, $S_{j,\alpha}$ commits to it by broadcasting $\mathcal{O}(\mathsf{seed}_{j,\alpha} \| s_{j,\alpha} \| j \| \alpha)$.

After the random seeds are committed, each $\Pi_{\mathsf{Gb}}(V_j)$ is run via $\mathcal{F}_{\mathsf{VP2P}}$ to make all the messages committed. In particular, if a server S_i emulates $S_{j_1,\alpha_1}, S_{j_2,\alpha_2}$ and a server S_j emulates $S_{j_1,\beta_1}, S_{j_2,\beta_2}$. The messages sent from S_{j_1,α_1} to S_{j_1,β_1} during $\Pi_{\mathsf{Gb}}(V_{j_1})$ in each round are sent together with a single message index mid_1 via $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j) = \mathcal{F}_{\mathsf{VP2P}}(S_{j_1,\alpha_1}, S_{j_1,\beta_1})$, while the messages from S_{j_2,α_2} to S_{j_2,β_2} in the same round during $\Pi_{\mathsf{Gb}}(V_{j_2})$ are sent with another index $\mathsf{mid}_2 \neq \mathsf{mid}_1$. Finally, the servers $S_{j,1}, \ldots, S_{j,c}$ commit to their outputs from $\Pi_{\mathsf{Gb}}(V_j)$. For each server $S_{j,\alpha}$'s output $\mathsf{gbout}_{j,\alpha}$ of $\Pi_{\mathsf{Gb}}(V_j)$, $S_{j,\alpha}$ samples a κ-bit random string $r_{j,\alpha}$ and broadcasts $\mathcal{O}(\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$ as a commitment.

6.5 Verification of Local Computations

Then, we introduce how the parties' local computations are checked. Specifically, the check follows the MPC-in-the-head framework [IKOS07, IPS08]. We let all servers agree on a random coin via $\mathcal{F}_{\mathsf{Coin}}$ that determines a set $\mathcal{T}_{\mathsf{ver}}$ of $N/240$ virtual servers in the first execution of Π_{GLOS} (resp. $N'/240$ virtual servers in the second execution). Then, the servers open their committed views during the emulation of these virtual servers to all the parties, and the parties can check whether each party's local computation is correctly performed. The protocol Π_{CompVer} for the check is given in Section I.

Recall that for any set of 3κ virtual servers emulated by a server S, a randomly chosen set of $N/240 = n\kappa$ virtual servers in the first execution of Π_{GLOS} (or $N'/240 = n'\kappa$ virtual servers in the first execution) includes a virtual server among the 3κ virtual servers with an overwhelming probability (see Section 4.1). Thus, each server S that remains uneliminated after the check should follow the protocol to perform the computation on at least all but 3κ virtual servers (with an overwhelming probability). Let all the active virtual servers, i.e., the virtual servers emulated completely by the remaining servers that are not eliminated, form a set $\mathcal{T}_{\mathsf{act}}$. Let those honest active virtual parties $V_j \notin \mathcal{T}_{\mathsf{ver}}$ such that $S_{j,1}, \ldots, S_{j,c}$ all honestly follow the protocol in Π_{ShVer} and $\Pi_{\mathsf{Gb}}(V_j)$ form the set $\mathcal{T}_{\mathsf{hon}}$. Π_{ShVer} guarantees the third property of $\mathcal{T}_{\mathsf{hon}}$, and Π_{CompVer} guarantees the second and fourth properties of $\mathcal{T}_{\mathsf{hon}}$ (see Section 4.3 for the properties). Therefore:

- In the first execution of Π_{GLOS}, the number of virtual servers in $\mathcal{T}_{\mathsf{act}} \backslash (\mathcal{T}_{\mathsf{hon}} \cup \mathcal{T}_{\mathsf{ver}})$ should be at most $3\kappa t < N/160$. Also recall that $T = N/30$, $|\mathcal{T}_{\mathsf{cor}}| < N/60$ (see Section 4.1), and $|\mathcal{T}_{\mathsf{ver}}| = N/240$, so we have $|\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}| - |\mathcal{T}_{\mathsf{hon}}| \leq N/160 + N/240 + N/60 < T$.
- In the second execution of Π_{GLOS}, the number of virtual servers in $\mathcal{T}_{\mathsf{act}} \backslash (\mathcal{T}_{\mathsf{hon}} \cup \mathcal{T}_{\mathsf{ver}})$ should be at most $3\kappa t/2 < N'/240$. Also recall that $T' = N'/60$, $|\mathcal{T}_{\mathsf{cor}}| < N'/120$ (see Section 4.1), and $|\mathcal{T}_{\mathsf{ver}}| = N'/240$, so we have $|\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}| - |\mathcal{T}_{\mathsf{hon}}| \leq N'/240 + N'/240 + N'/120 \leq T'$.

Thus, $\mathcal{T}_{\mathsf{hon}}$ satisfies all the properties listed in Section 4.3. After the verification, sending all the outputs of $\Pi_{\mathsf{Gb}}(V_j)$ for $V_j \in \mathcal{T}_{\mathsf{act}}$ to a corrupted client is secure, so we can let each client collect these outputs for the evaluation.

6.6 Collecting the Outputs from the Garbling Phase

After the verification, the clients need to collect the outputs from $\Pi_{\mathsf{Gb}}(V_j)$ for each virtual server $V_j \in \mathcal{T}_{\mathsf{act}}$. For this task, we reuse the committees for virtual servers. For each server $S_{j,\alpha}$ that emulates a remaining virtual server V_j, $S_{j,1}, \ldots, S_{j,c}$ open their committed outputs of $\Pi_{\mathsf{Gb}}(V_j)$ to $S_{j,\alpha}$. Each $S_{j,\alpha}$ then checks whether the outputs from them are correctly opened. If not, $S_{j,\alpha}$ accuses the corrupted server as in the instantiation of the verifiable P2P channel to make the corrupted server either eliminated or send the correct message to him. After that, if the outputs of $\Pi_{\mathsf{Gb}}(V_j)$ are collected, each $S_{j,\alpha}$ sends the outputs to all the parties. We provide the detailed protocol Π_{Collect} as follows in Figure 7.

Protocol Π_{Collect}

For each active virtual server V_j, all the servers hold the commitment $\mathcal{O}(\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$ for each $\alpha = 1, \ldots, c$, where $\mathsf{gbout}_{j,\alpha}$ is $S_{j,\alpha}$'s output from $\Pi_{\mathsf{Gb}}(V_j)$. Then:

1. Each server $S_{j,\alpha}$ sends $\mathsf{gbout}_{j,\alpha}, r_{j,\alpha}$ to $S_{j,1}, \ldots, S_{j,c}$.
2. Each server $S_{j,\alpha}$ checks whether $\mathcal{O}(\mathsf{gbout}_{j,\beta} \| r_{j,\beta} \| \mathsf{mid}_{\mathsf{gbout}_{j,\beta}})$ matches the commitment for $S_{j,\beta}$'s output for each $\beta = 1, \ldots, c$. If not, $S_{j,\alpha}$ broadcasts $(\mathsf{complain}, \mathsf{mid}_{\mathsf{gbout}_{j,\beta}}, S_{j,\alpha}, S_{j,\beta})$. Otherwise, $S_{j,\alpha}$ sends $\mathsf{gbout}_{j,\beta}, r_{j,\beta}$ to all the clients.
3. If $(\mathsf{complain}, \mathsf{mid}, S_{j,\alpha}, S_{j,\beta})$ is received:
   (a) $S_{j,\beta}$ sends $\mathsf{gbout}_{j,\beta}, r_{j,\beta}$ to all the servers.
   (b) Each server checks whether $\mathcal{O}(\mathsf{gbout}_{j,\beta} \| r_{j,\beta} \| \mathsf{mid}_{\mathsf{gbout}_{j,\beta}})$ matches the commitment for $S_{j,\beta}$'s output. If not, the server broadcasts $(\mathsf{eliminate}, S_{j,\beta})$.
   (c) If $(\mathsf{eliminate}, S_{j,\beta})$ is received from $\geq t+1$ servers, the parties mark $S_{j,\beta}$ "eliminated". Otherwise, the parties deliver $\mathsf{gbout}_{j,\beta}, r_{j,\beta}$ to $S_{j,\alpha}$. $S_{j,\alpha}$ then sends $\mathsf{gbout}_{j,\beta}, r_{j,\beta}$ to all the clients.

4. Each client P initializes a set $\mathcal{E}_P = \emptyset$. If none of $S_{j,1}, \ldots, S_{j,c}$ are marked "eliminated", each client P (who holds an output wire of C) checks whether $\mathcal{O}(\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$ matches the commitment for $S_{j,\alpha}$'s output for each $\alpha = 1, \ldots, c$. If all these outputs received from one of $S_{j,1}, \ldots, S_{j,c}$ match their commitments, P adds V_j into $\mathcal{E}_P$.

Figure 7: Protocol for collecting the outputs from the garbling phase.

After Π_{Collect} terminates, each client P runs $\Pi_{\mathsf{Ev}}(P)$ with the outputs from $\Pi_{\mathsf{Gb}}(V_j)$ from $V_j \in \mathcal{E}_P$ to compute his own output. Recall that we need to ensure that the subsharing of Σ with respect to $\mathcal{E}_P$ has the property of 3-multiplicative e-error-correction for the correctness of the evaluation output. We analyze this in two executions of Π_{GLOS} respectively:

- For the first execution of Π_{GLOS}, $\Pi_{\mathsf{Ev}}(P)$ for an honest client P is only run when fewer than $t/2$ servers are eliminated. In this case, our parameter choice ensures that the number of active honest virtual servers is at least $N/6 = 40n\kappa$ (see Section 4.1). These virtual servers are guaranteed to be in $\mathcal{E}_P$ with an overwhelming probability. Thus, for the subsharing of (N, T, k, ℓ)-LSSS Σ with respect to $\mathcal{E}_P$, we have $|\mathcal{E}_P| \geq N/6$, $T = N/30$, and $e \leq |\mathcal{T}_{\mathsf{act}} \backslash (\mathcal{T}_{\mathsf{hon}} \cup \mathcal{T}_{\mathsf{ver}})| \leq N/160$[9]. It holds that $T \leq (|\mathcal{E}_P| - 2e)/4$, so the subsharing of Σ with respect to $\mathcal{E}_P$ has the property of 3-multiplicative e-error-correction.
- For the second execution of Π_{GLOS}, the number of virtual servers completely emulated by honest servers is at least $N'/12 = 20n'\kappa$ (see Section 4.1). For the subsharing of (N', T', k', ℓ')-LSSS Σ with respect to $\mathcal{E}_P$, we have $|\mathcal{E}_P| \geq N'/12$, $T' = N'/60$, and $e \leq |\mathcal{T}_{\mathsf{act}} \backslash (\mathcal{T}_{\mathsf{hon}} \cup \mathcal{T}_{\mathsf{ver}})| \leq N'/240$. It holds that $T' \leq (|\mathcal{E}_P| - 2e)/4$, so the subsharing of Σ with respect to $\mathcal{E}_P$ still has the property of 3-multiplicative e-error-correction.

Therefore, each honest client P can correctly evaluate his output using $\Pi_{\mathsf{Ev}}(P)$.

7 Main Protocol

Finally, we put everything together and present our main protocol Π_{Main}. Let $\mathcal{O}$ be a random oracle with a sufficiently large (but still $\mathsf{poly}(n, \kappa)$) output length.

Protocol Π_{Main}

The protocol runs among $\tilde{n}$ clients and n servers in computing a circuit C. All the servers initialize $\mathsf{Rerun} = 0$, and $\mathcal{P}_S$ to be the set of all servers at the beginning. During the protocol execution, once a server is marked "eliminated", it is deleted from $\mathcal{P}_S$. If $\mathsf{Rerun} = 0$, once there are $\geq t/2$ servers marked "eliminated", all the parties set $\mathsf{Rerun} = 1$ and rerun the protocol from Step 2. The parties do not interact with a party P if P is marked "eliminated".

1. **VSS Inputs.** For each input wire w attached to a client $\mathtt{C}_i$, $\mathtt{C}_i$ serves as the dealer and sends x to $\mathcal{F}_{\mathsf{VSS}}$. All the parties send VSS to $\mathcal{F}_{\mathsf{VSS}}$, where x is the input value for w. The parties agree on the circuit C' that first runs $\mathsf{VSS.Rec}$ on all the VSS shares of inputs and then computes the circuit C, where the servers serve as n clients that provide their VSS shares as inputs to C', and the $\tilde{n}$ real clients serve as another $\tilde{n}$ clients that only receive outputs from C'.
2. **Determining Virtual Servers.** Let $c = 6$. All the servers call $\mathcal{F}_{\mathsf{Coin}}$ with $\mathsf{RandCoin}$ and obtain a random κ-bit string r_{vp}.
   - If $\mathsf{Rerun} = 0$, let $N = 240n\kappa$. Then, the servers then compute $\mathcal{O}(r_{\mathsf{vp}})$ to get c-server subsets $\{S_{j,1}, \ldots, S_{j,c}\}$ of $\mathcal{P}_S$ for $j = 1, \ldots, N$.
   - If $\mathsf{Rerun} = 1$, let n' be the number of servers in $\mathcal{P}_S$ and let $N' = 240n'\kappa$. Then, the servers then compute $\mathcal{O}(r_{\mathsf{vp}})$ to get c-server subsets $\{S_{j,1}, \ldots, S_{j,c}\}$ of $\mathcal{P}_S$ for $j = 1, \ldots, N'$.

   The servers $S_{j,1}, \ldots, S_{j,c}$ will emulate a virtual server V_j in the first execution of Π_{GLOS}. A virtual server V_j is called an active one if none of $S_{j,1}, \ldots, S_{j,c}$ are marked "eliminated".

[9] Here, $\mathcal{T}_{\mathsf{act}}$ and $\mathcal{T}_{\mathsf{hon}}$ are updated after Π_{Collect}. The virtual servers that are no longer active are removed from the sets.

3. **Generating Sharings.** The servers run Π_{Share} (introduced in Section 4.1), where when each server S send the shares for a server $S_{j,\alpha}$ emulating V_j, he sends the shares as a single message via $\mathcal{F}_{\mathsf{VP2P}}(S, S_{j,\alpha})$. During the execution, if $(\mathsf{eliminate}, S_i)$ is received, the parties mark S_i "$\mathsf{eliminated}$". After Π_{Share}, for each message with index mid that is sent between two remaining servers in $\mathcal{P}_S$, $(\mathsf{committed}, \mathsf{mid})$ is received by all parties.
4. **Generating Auxiliary Sharings.** Let $\kappa' = N + \kappa$. Each party P generates a random $\Sigma_{\times\kappa'}$-sharing $[\boldsymbol{r}_1^P]_{\times\kappa'}$, a random $\Sigma^{(2)}_{\times\kappa'}$-sharing $[\boldsymbol{r}_2^P]^{(2)}_{\times\kappa'}$, a random $\Sigma^{(3)}_{\times\kappa'}$-sharing $[\boldsymbol{r}_3^P]^{(3)}_{\times\kappa'}$, and a random $\Sigma^{(2)}_{\times\kappa'}$-sharing $[\boldsymbol{o}^P]^{(2)}_{\times\kappa'}$ with an all-zero secret. Then, P distributes them to the virtual servers, where the shares for a server $S_{j,\alpha}$ are sent together via $\mathcal{F}_{\mathsf{VP2P}}(P, S_{j,\alpha})$. During the execution, if $(\mathsf{eliminate}, S_i)$ is received, the parties mark S_i "$\mathsf{eliminated}$".
5. **Preparing Base ROTs.** All the servers send ROT to $\mathcal{F}_{\mathsf{BaseROT}}$ and receive the pair-wise base ROT correlations and their VSS shares of them from $\mathcal{F}_{\mathsf{BaseROT}}$.
6. **Verifying the Sharings.** The parties run Π_{ShVer} (Figure 6) to check the sharings generated by each server during Π_{Share}.
7. **Committing to the Randomness.** Each server $S_{j,\alpha}$ who emulates a virtual server V_j samples the random seed $\mathsf{seed}_{j,\alpha} \in \mathbb{F}_2^\kappa$ as his randomness to $\Pi_{\mathsf{Gb}}(V_j)$ (introduced in Section 4.2) and another κ-bit string $s_{j,\alpha}$. Then, $S_{j,\alpha}$ commits to $\mathsf{seed}_{j,\alpha}$ by broadcasting $\mathcal{O}(\mathsf{seed}_{j,\alpha} \| s_{j,\alpha} \| j \| \alpha)$.
8. **Garbling the Computations.** For each active virtual server V_j, the servers $S_{j,1}, \ldots, S_{j,c}$ run $\Pi_{\mathsf{Gb}}(V_j)$ with the shares obtained from Π_{Share} and the random seeds sampled in the last step. Each message is sent via $\mathcal{F}_{\mathsf{VP2P}}$. During the execution, if $(\mathsf{eliminate}, S_i)$ is received, the parties mark S_i "$\mathsf{eliminated}$". After $\Pi_{\mathsf{Gb}}(V_j)$ terminates, for each message with index mid that is sent between two remaining servers in $\mathcal{P}_S$, $(\mathsf{committed}, \mathsf{mid})$ is received by all parties. Then, for each server $S_{j,\alpha}$'s output $\mathsf{gbout}_{j,\alpha}$ of $\Pi_{\mathsf{Gb}}(V_j)$, $S_{j,\alpha}$ samples a κ-bit random string $r_{j,\alpha}$ and broadcasts $\mathcal{O}(\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$.
9. **Verifying the Computations.** The parties run Π_{CompVer} (Figure 24) to check the servers' local computations.
10. **Collecting Garbled Circuits.** The parties run Π_{Collect} (Figure 7) to let the clients collect the garbled circuits.
11. **Evaluating Garbled Circuits.** Each client P runs $\Pi_{\mathsf{Ev}}(P)$ (in Section 4.4) to get his own outputs from C', which is the same with his outputs from C.

Figure 8: The main protocol.

Theorem 2 *When $n = 2t + 1$, the main protocol Π_{Main} is a computationally secure MPC protocol in the $\{\mathcal{F}_{\mathsf{VSS}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{VP2P}}, \mathcal{F}_{\mathsf{BaseROT}}\}$-hybrid model against a PPT adversary $\mathcal{A}$ who corrupts any number of clients and up to t servers.*

Replacing the functionalities with their instantiation in the plain/RO model, the achieved communication cost of Π_{Main} is $O(|C|(n+\tilde{n})\kappa + D\cdot(n+\tilde{n})n^2\kappa^3 + W_I \cdot \mathsf{poly}(n, \tilde{n}, \kappa))$ bits via P2P channels and $W_I \cdot \mathsf{poly}(n, \kappa)$ bits via broadcast channels. We provide a detailed security proof in Section J and a cost analysis in Section K. Finally, we take $\tilde{n} = n$ in the main protocol and let each of the n parties emulate both a client and a server. Then, the protocol is transferred into the standard model, where the communication cost via P2P channels becomes $O(|C|n\kappa + Dn^3\kappa^3 + W_I \cdot \mathsf{poly}(n, \kappa))$ bits.

Note that the communication cost via broadcast channels is related to the input size, which is not ideal. We can use the following way to do the broadcasts in each round to reduce the communication cost via broadcast channels to $\mathsf{poly}(n, \kappa)$:

1. For each message m broadcast by a party P, P samples a random κ-bit string r and broadcasts $\mathcal{O}(m \| r \| \mathsf{mid}_m)$ as a commitment for m.
2. Then, P sends m, r to all the parties via P2P channels.
3. Each party checks the commitment. If the check fails, the party broadcasts $(\mathsf{fail}, \mathsf{mid}_m)$.

4. If at least $t+1$ parties' broadcasts of $(\mathsf{fail}, \mathsf{mid}_m)$ are received, the parties set $m = \perp$. Otherwise, the parties whose checks pass send m, r to all the parties. Finally, each party checks the commitment and obtains the correct message m.

In this way, broadcasting each message m only requires $O(|m|n^2 + \mathsf{poly}(n,\kappa))$-bit communication via P2P channels and $\mathsf{poly}(n,\kappa)$-bit communication via broadcast channels. Using this trick, we can reduce the communication cost via broadcast channels to $\mathsf{poly}(n,\kappa)$ bits, while the communication cost via P2P channels remains $O(|C|n\kappa + Dn^3\kappa^3 + W_I \cdot \mathsf{poly}(n,\kappa))$ bits. Finally, we restate our main theorem.

Theorem 1 *Assuming broadcast channels and a random oracle, there is a computationally secure constant-round MPC protocol with GOD against a fully malicious adversary controlling up to $t = (n-1)/2$ parties with communication complexity $O(|C|n\kappa + Dn^3\kappa^3 + W_I \cdot \mathsf{poly}(n,\kappa))$ bits via P2P channels and $\mathsf{poly}(n,\kappa)$ bits via broadcast channels, where $|C|$ is the circuit size, D is the circuit depth, W_I is the number of input wires of C, and κ is the computational security parameter.*

References

[AAPP23] Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. Detect, pack and batch: Perfectly-secure MPC with linear communication and constant expected time. In *Advances in Cryptology – EUROCRYPT 2023, Part II*, volume 14005 of *Lecture Notes in Computer Science*, pages 251–281. Springer, 2023.

[AAY21] Ittai Abraham, Gilad Asharov, and Avishay Yanai. Efficient perfectly secure computation with optimal resilience. In *TCC 2021: 19th Theory of Cryptography Conference, Part II*, volume 13043 of *Lecture Notes in Computer Science*, pages 66–96. Springer, 2021.

[ABDE24] Amit Agarwal, Alexander Bienstock, Ivan Damgård, and Daniel Escudero. Honest majority GOD MPC with O(depth(C)) rounds and low online communication. In *Advances in Cryptology – ASIACRYPT 2024, Part VI*, volume 15489 of *Lecture Notes in Computer Science*, pages 234–265. Springer, 2024.

[ABT21] Benny Applebaum, Zvika Brakerski, and Rotem Tsabary. Perfect secure computation in two rounds. *SIAM J. Comput.*, 50(1):68–97, January 2021.

[ACGJ18] Prabhanjan Ananth, Arka Rai Choudhuri, Aarushi Goel, and Abhishek Jain. Round-optimal secure multiparty computation with honest majority. In *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 395–424. Springer, 2018.

[AJL+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 483–501. Springer, 2012.

[AKP22] Benny Applebaum, Eliran Kachlon, and Arpita Patra. Round-optimal honest-majority MPC in minicrypt and with everlasting security - (extended abstract). In *TCC 2022: 20th Theory of Cryptography Conference, Part II*, volume 13748 of *Lecture Notes in Computer Science*, pages 103–120. Springer, 2022.

[AKP23] Benny Applebaum, Eliran Kachlon, and Arpita Patra. The round complexity of statistical MPC with optimal resiliency. In *55th Annual ACM Symposium on Theory of Computing*, pages 1527–1536. ACM Press, 2023.

[Bea89] Donald Beaver. Multiparty protocols tolerating half faulty processors. In *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 560–572. Springer, 1989.

[BFO12] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 663–680, 2012.

[BGH+23] Gabrielle Beck, Aarushi Goel, Aditya Hegde, Abhishek Jain, Zhengzhong Jin, and Gabriel Kaptchuk. Scalable multiparty garbling. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023*, pages 2158–2172. ACM, 2023.

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

[BJMS20] Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. Secure MPC: Laziness leads to GOD. In *Advances in Cryptology – ASIACRYPT 2020, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 120–150. Springer, 2020.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *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.

[BTH06] Zuzana Beerliová-Trubíniová and Martin Hirt. Efficient multi-party computation with dispute control. In *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 305–328. Springer, 2006.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In *TCC 2008: 5th Theory of Cryptography Conference*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230. Springer, 2008.

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

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

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

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 11–19, New York, NY, USA, 1988. Association for Computing Machinery.

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

[CDD+99] Ronald Cramer, Ivan Damgård, Stefan Dziembowski, Martin Hirt, and Tal Rabin. Efficient multiparty computations secure against an adaptive adversary. In *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 311–326, 1999.

[CGHZ16] Sandro Coretti, Juan A. Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multi-party computation based on one-way functions. In *Advances in Cryptology – ASIACRYPT 2016, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 998–1021. Springer, 2016.

[Cle86] R Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, page 364–369, New York, NY, USA, 1986. Association for Computing Machinery.

[CP17] Ashish Choudhury and Arpita Patra. An efficient framework for unconditionally secure multiparty computation. *IEEE Transactions on Information Theory*, 63(1):428–468, 2017.

[DI05] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In *Advances in Cryptology - CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 378–394. Springer, 2005.

[DIK+08] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience. In *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 241–261. Springer, 2008.

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

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *24th Annual ACM Symposium on Theory of Computing*, pages 699–710. ACM Press, 1992.

[GIP+14] Daniel Genkin, Yuval Ishai, Manoj M. Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '14, page 495–504, New York, NY, USA, 2014. Association for Computing Machinery.

[GLM+24] Vipul Goyal, Junru Li, Ankit Kumar Misra, Rafail Ostrovsky, Yifan Song, and Chenkai Weng. Dishonest majority constant-round mpc with linear communication from ddh. In *Advances in Cryptology – ASIACRYPT 2024*, pages 167–199, 2024.

[GLOS25] Vipul Goyal, Junru Li, Rafail Ostrovsky, and Yifan Song. Towards building scalable constant-round MPC from minimal assumptions via round collapsing. In *Advances in Cryptology – CRYPTO 2025, Part IV*, volume 16003 of *Lecture Notes in Computer Science*, pages 70–104, Santa Barbara, CA, USA, August 17–21, 2025. Springer, Cham, Switzerland.

[GLS15] S. Dov Gordon, Feng-Hao Liu, and Elaine Shi. Constant-round MPC with fairness and guarantee of output delivery. In *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 63–82. Springer, 2015.

[GLS19] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In *Advances in Cryptology – CRYPTO 2019, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 85–114, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Cham, Switzerland.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *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 *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference*, 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 *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference*, volume 13510 of *Lecture Notes in Computer Science*, pages 3–32. Springer, 2022.

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 618–646. Springer, 2020.

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

[HMP00] Martin Hirt, Ueli M. Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In Tatsuaki Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 143–161. Springer, 2000.

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

[Imp95] R. Impagliazzo. A personal view of average-case complexity. In *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*, pages 134–147, 1995.

[IOZ14] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *Advances in Cryptology – CRYPTO 2014, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 369–386. Springer, 2014.

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

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

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[LS25] Junru Li and Yifan Song. Honest majority constant-round MPC with linear communication from one-way functions. In *TCC 2025: 23rd Theory of Cryptography Conference, Part I*, volume 16268 of *Lecture Notes in Computer Science*, pages 3–32, Aarhus, Denmark, December 1–5, 2025. Springer, Cham, Switzerland.

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

[Roy22] Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part I*, volume 13507 of *Lecture Notes in Computer Science*, pages 657–687, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Cham, Switzerland.

[SASM10] Takenobu Seito, Tadashi Aikawa, Junji Shikata, and Tsutomu Matsumoto. Information-theoretically secure key-insulated multireceiver authentication codes. In *AFRICACRYPT 10: 3rd International Conference on Cryptology in Africa*, volume 6055 of *Lecture Notes in Computer Science*, pages 148–165, 2010.

[SHZI02] Junji Shikata, Goichiro Hanaoka, Yuliang Zheng, and Hideki Imai. Security notions for unconditionally secure signature schemes. In *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 434–449, 2002.

[SS11] Colleen Swanson and Douglas R. Stinson. Unconditionally secure signature schemes revisited. In *ICITS 11: 5th International Conference on Information Theoretic Security*, volume 6673 of *Lecture Notes in Computer Science*, pages 100–116, 2011.

[WB86] Lloyd R Welch and Elwyn R Berlekamp. Error correction for algebraic block codes, December 30 1986. US Patent 4,633,470.

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *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.

[YWZ20] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1627–1646. ACM, 2020.

A An Overview for [GLOS25]

In this section, we give an overview of how the work [GLOS25] achieves a constant-size ($O(|C|\kappa + D \cdot \mathsf{poly}(n, \kappa))$) multiparty garbled circuit. The main idea is to garble the next-message functions of an underlying protocol, ensuring that the outputs of the garbled circuits in each round are sufficient for computing the input labels for the garbled circuits of the next round.

The Underlying Protocol. We first introduce the structure of the underlying protocol compiled in [GLOS25]. In particular, the protocol is constructed in the setting of $n > 4t$ and is based on a secret sharing scheme Σ (denoted by $[\cdot]$) with a secret size $k = \Theta(n)$ and a constant share size [CC06]. Such a secret sharings scheme is linear over $\mathbb{F}_2$. Like the standard packed Shamir secret sharing scheme [FY92], the local tensor products of the shares of two such sharings still form a similar linear secret sharing $\Sigma^{(2)}$ (denoted by $[\cdot]^{(2)}$, similar for $[\cdot]^{(3)}$ that denotes the product of a $\Sigma^{(2)}$-sharing and a Σ-sharing), where the only difference is that we need more shares for reconstruction. We denote the product of For simplicity, we introduce the underlying protocol for a SIMD circuit that computes k^2 copies of the same circuit together. Such a protocol can be generated to support general circuits using the network routing technique [GPS21, GPS22].

In general, the protocol is constructed in a gate-by-gate manner. For each group of addition/multiplication gates (over $\mathbb{F}_2$), 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. 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, utilizing the multiplicative property of Σ, $[\boldsymbol{z}]^{(2)} = [\boldsymbol{x}] * [\boldsymbol{y}]$ can be locally computed. Now it is sufficient to transform $[\boldsymbol{z}]^{(2)}$ to $[\boldsymbol{z}]$.

To perform this transformation, the idea of [GLOS25] is to transform a batch of k $\Sigma^{(2)}$-sharings each time. Suppose the input sharings are denoted by $[\boldsymbol{z}_1]^{(2)}, \ldots, [\boldsymbol{z}_k]^{(2)}$. The secrets can be viewed 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 $\boldsymbol{Z}$. The solution is to first let all parties obtain $[\boldsymbol{Z}_{\star,1}], \ldots, [\boldsymbol{Z}_{\star,n}]$ only using reconstruction of $\Sigma^{(2)}$-sharings. Since the sharing algorithm $\Sigma.\mathsf{Sh}$ of Σ is $\mathbb{F}_2$-linear, we may view it 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 $[\boldsymbol{Z}_{\star,1}], \ldots, [\boldsymbol{Z}_{\star,k}]$.

To complete the transformation and obtain $[\boldsymbol{z}_1], \ldots, [\boldsymbol{z}_k]$, we only need to apply the above process on $[\boldsymbol{Z}_{\star,1}], \ldots, [\boldsymbol{Z}_{\star,k}]$ again.

As demonstrated, the evaluation phase contains only reconstructions of $\Sigma^{(2)}$-sharings, and each party's local computations depend exclusively on the secrets contained in these sharings and the preprocessed/input data. Furthermore, the $\Sigma^{(2)}$-sharings to be reconstructed are public sharings that can be known by all the parties. This is an important property utilized by [GLOS25]. In particular, the above protocol for a general circuit C only requires communication of $O(|C| + D \cdot \mathsf{poly}(n))$ bits and $O(|C| + D \cdot \mathsf{poly}(n))$ AND operations in the parties' local computations.

Round Collapsing via Garbling Local Computations. The starting point of [GLOS25] is to garble the local computations of all the parties during the evaluation phase of the underlying protocol and send the garbled circuits to an evaluator. However, merely possessing the garbled circuit is insufficient. The evaluator should also be able to input wire labels for the secrets of the $\Sigma^{(2)}$-sharings $[\boldsymbol{z}]^{(2)}$ that need to be reconstructed during the protocol.

The core technique for preparing the wire labels $k_{w,0}$ and $k_{w,1}$ (corresponding to wire w carrying each bit of the secret $\boldsymbol{z}$) involves the following initial setup: the evaluator distributes $[\boldsymbol{k_0}]^{(3)}$ and $[\boldsymbol{k_1} - \boldsymbol{k_0}]$ (where

$\boldsymbol{k_0}$ is the vector containing $k_{w,0}$ for the wires carring all the bits of $\boldsymbol{z}$) during the preprocessing phase. Given $[\boldsymbol{z}]^{(2)}$, parties can then locally compute the label's sharing via:

$$[\boldsymbol{k_z}]^{(3)} = [\boldsymbol{k_0}]^{(3)} + [\boldsymbol{z}]^{(2)} \otimes [\boldsymbol{k_1} - \boldsymbol{k_0}].$$

Crucially, recall that in the underlying protocol, all reconstructed $\Sigma^{(2)}$-sharings are public. Consequently, the evaluator learns these sharings $[\boldsymbol{z}]^{(2)}$ during the garbled circuit evaluation. This enables the evaluator to reconstruct both the underlying secret $\boldsymbol{z}$ and its corresponding wire label $\boldsymbol{k_z}$. With this information, the evaluator can simulate the entire execution of the underlying protocol.

Now, let's analyze the communication cost of the above approach. For each reconstruction in the underlying protocol, the receiver needs to distribute two sharings for the label (whose size is $O(\kappa)$) of the secret. The communication cost of these steps is close to κ times the communication cost of the underlying protocol. Given that the communication cost of protocol is $O(|C| + D \cdot \mathsf{poly}(n))$ (as we need to reconstruct $[\boldsymbol{z}]^{(2)}$ to all the parties for each multiplication gate), the communication cost of distributing these sharings is $O(|C|\kappa + D \cdot \mathsf{poly}(n, \kappa))$. The other cost mainly comes from sending the garbled circuit. Since we can apply the FreeXOR technique [KS08] in the random oracle model, it only relies on the number of local AND operations in the underlying protocol. Given that the number of local AND operations in the underlying protocol is also $O(|C| + D \cdot \mathsf{poly}(n))$, the total garbled circuit size is $O(|C|\kappa + D \cdot \mathsf{poly}(n, \kappa))$ using FreeXOR [KS08].

Boosting the Corruption Threshold via Party Virtualization. Note that the above approach requires $n = 4t$ in the underlying protocol. To boost the corruption threshold to standard honest majority $n = 2t + 1$, the authors of [GLOS25] use the party virtualization technique [Bra87]. The core idea is to have randomly selected committees of c parties emulate each virtual party executing the underlying protocol. A dishonest majority MPC protocol (specifically, a BMR-type multiparty garbling process, which ensures black-box use of cryptographic primitives) is used to garble the local computation assigned to each virtual party. Crucially, all data associated with a virtual party (e.g., inputs, randomness, intermediate states) is additively shared among its c-party committee. This structure limits adversarial access: the adversary learns the internal data of a virtual party only if all c parties emulating it are corrupted. Consequently, the fraction of compromised virtual parties is approximately 2^{-c}, which becomes negligible for sufficiently large c.

To achieve security with abort against a malicious adversary, the authors of [GLOS25] apply the "Watchlist" method [IKOS07, IPS08]. In more detail, the parties commit their inputs and randomness before they participate in the inner protocol. After the virtual parties complete their garbling, each party will choose a public set of virtual parties. The parties emulating it will open their view (during the emulation of the virtual party) to the party. This ensures that any malicious deviation affecting a constant fraction of virtual parties will be detected with overwhelming probability, triggering protocol abort.

B Additional Preliminaries

In this section, we provide additional preliminaries for our protocol and proof.

B.1 Security Model

UC Model. We prove our constructions secure in the UC framework [Can01]. Protocol machines, ideal functionalities, the adversary, and the environment $\mathcal{Z}$ are all modeled as interactive Turing machine (ITM) instances. 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 receive 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 $\tilde{n}$ 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 any number of clients and t servers. In this work, we consider a fully malicious adversary, where the corrupted clients and servers can deviate from the protocol arbitrarily.

Real-World Execution. In the real world, an execution of protocol Π consists of a series of activations of the ITM instances, starting with the environment $\mathcal{Z}$ that provides inputs to and collects outputs from the parties and the adversary $\mathcal{A}$, parties can also give input to and collect output from sub-parties (e.g., hybrid functionalities), and $\mathcal{A}$ can communicate with parties via "backdoor" messages. At the end of the protocol, the output of the real-world execution is the random variable containing the output of $\mathcal{Z}$ with input z, secure parameter κ, and interacting with the clients, servers, and the adversary $\mathcal{A}$ with certain random tapes.

Ideal-World Execution. The ideal world contains $\tilde{n}$ dummy clients, n dummy servers, an ideal-world adversary Sim (the simulator), an environment $\mathcal{Z}$, and an ideal functionality $\mathcal{F}$ (the trusted party). The environment gives inputs to the honest parties, receives outputs, and also interacts with the ideal adversary. The dummy clients and servers just forward inputs from $\mathcal{Z}$ to $\mathcal{F}$ and forward outputs from $\mathcal{F}$ to $\mathcal{Z}$. 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 $\mathcal{F}$.

We say that a protocol t-securely realizes $\mathcal{F}$ if for any PPT adversary $\mathcal{A}$ that controls at most t servers, there exists a simulator Sim such that for any PPT environment $\mathcal{Z}$, the distribution of the output of the real-world execution is *computationally indistinguishable* from the distribution of the output of the ideal-world execution. We refer the readers to [Can01] for more details.

B.2 Linear Secret Sharing Schemes.

We first introduce the formal definitions of linear secret sharing schemes (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 [GLOS25]) *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 $\le t$,*

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

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

We then define the multiplication property for an LSSS.

Definition 3 *We say an (n, t, k, ℓ)-LSSS Σ has d-multiplicative e-error-correction 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 $j = 3, \ldots, d$.*

- *Any $n-2e$ shares of an $\Sigma^{(d)}$-sharing uniquely determine the secret of the $\Sigma^{(d)}$-sharing, i.e., the secret can be reconstructed from a $\Sigma^{(d)}$-sharing when e shares of it are incorrect.*

In this work, we need to handle a set of shares of an LSSS Σ, and these shares also need to form an LSSS. We call this kind of sharing a subsharing of Σ.

Definition 4 *We call an (n,t,k,ℓ)-LSSS $\tilde{\Sigma}$ a subsharing of an (N,t,k,ℓ)-LSSS Σ with respect to a set S with $|S|=n, S\subset\{1,\dots,N\}$ if*

$$\tilde{\Sigma}.\mathsf{Sh}(\cdot,\cdot)=\pi_S(\Sigma.\mathsf{Sh}(\cdot,\cdot)).$$

In particular, for an (n,t,k,ℓ)-subsharing for the LSSS Σ given by Lemma 1, the corresponding subsharing of $\Sigma^{(3)}$ supports an error correction against up to e errors in polynomial time using Berlekamp–Welch algorithm [WB86].

Remark 1 *As noted in [GLOS25], for an (n,t,k,ℓ)-LSSS Σ, we have the following algorithms. There are algorithms that can efficiently determine whether a set S of shares comes from a valid Σ-sharing and give such a sharing (if it is valid). There are also algorithms that can efficiently sample a set of at most t_1 shares from another t_2 share ($t_1+t_2\le t$) of a Σ-sharing without the secret. We refer the readers to [GLOS25] for more details.*

B.3 Verifiable Secret Sharings

Verifiable Secret Sharing (VSS) [BGW88, RB89] enables a dealer to generate a sharing for a secret s to n parties such that any t parties' views are independent of the secret, where there is a reconstruction algorithm to reconstruct the secret from all the parties' views even if t parties do not send their view correctly. A VSS is modeled as two algorithms $(\mathsf{VSS.Sh},\mathsf{VSS.Rec})$, where

- $\mathsf{VSS.Sh}$ is an $\mathsf{poly}(n,\kappa)$-time randomized algorithm (the space for the randomness is denoted by $\mathcal{R}$, which includes a set of t shares in the VSS), and the input is a bit $s\in\{0,1\}$. $\mathsf{VSS.Sh}(s;r)$ outputs a vector $\boldsymbol{c}=(c_1,\dots,c_n)\in(\mathbb{F}_2^\ell)^n$, where for all $\boldsymbol{s},\boldsymbol{s}'\in\{0,1\}$ and $A\subset\{1,\dots,n\}$ of size $\le t$,

$$\{r\xleftarrow{\$}\mathcal{R},\boldsymbol{c}\leftarrow\mathsf{VSS.Sh}(s,r):\pi_A(\boldsymbol{c})\}\equiv\{r'\xleftarrow{\$}\mathcal{R},\boldsymbol{c}'\leftarrow\mathsf{VSS.Sh}(s',r'):\pi_A(\boldsymbol{c}')\}.$$

- $\mathsf{VSS.Rec}$ is an $\mathsf{poly}(n,\kappa)$-time deterministic algorithm, and its input is a vector $\boldsymbol{c}=(c_1,\dots,c_n)\in(\mathbb{F}_2^\ell)^n$. It satisfies that for any $\boldsymbol{s}\in(\mathbb{F}_2^\ell)^n$, if $\exists A\subset\{1,\dots,n\},|A|=(n-t),s\in\{0,1\},r\in\mathcal{R}$ such that

$$\pi_A(\mathsf{VSS.Sh}(s,r))=\pi_A(\boldsymbol{c}),$$

  then $\mathsf{VSS.Rec}(\boldsymbol{c})=s$ satisfies with an overwhelming probability.

B.4 The Functionality $\mathcal{F}_{\mathsf{MPC}}$

We give the functionality $\mathcal{F}_{\mathsf{MPC}}$ here in Figure 9.

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

Upon receiving (MPC,C) (where C is an arithmetic circuit over $\mathbb{F}_{2^\kappa}$) from all the honest parties:

1. For each input wire w of C attached to P_i, the trusted party receives the input v_w on this wire from P_i.
2. Sim evaluates the circuit C with the inputs and obtains the wire value v_w for each output wire w of C. Then, the trusted party receives a set of corrupted parties' shares of $[\![v_w]\!]$ from Sim and samples the VSS $[\![v_w]\!]$ based on the secret and corrupted parties' shares. For each honest party P_i, Sim sends P_i's shares of $[\![v_w]\!]$ for each output wire w to P_i.

Figure 9: Functionality for a General MPC.

Lemma 6 ([CP17]) *$\mathcal{F}_{\mathsf{MPC}}$ can be t-securely realized with statistical security against a fully malicious adversary with communication of $O(|C|n^4)$ bits via both P2P channels and the broadcast channel in $O(\mathsf{depth}(C))$ rounds.*

C Chernoff Bound and Proof of Properties on Virtual Servers

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

Lemma 7 (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:

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

Next, we will use the Chernoff bound to show that when we choose $c = 6$, $N = 240n\kappa$, and $N' = 240n'\kappa$, we can ensure that:

1. With an overwhelming probability, the number of corrupted virtual servers is no more than $N/60 = 4n\kappa$ during the first execution of Π_{GLOS}, and the number of corrupted virtual servers is no more than $N'/120 = 2n'\kappa$ during the second execution of Π_{GLOS}.
2. During the first execution of Π_{GLOS}, for any set of $< t/2$ corrupted servers, with an overwhelming probability, the number of honest virtual servers completely emulated by the remaining $> n - t/2$ servers is at least $N/6 = 40n\kappa$.
3. During the second execution of Π_{GLOS}, with an overwhelming probability, the number of virtual servers completely emulated by honest servers is at least $N'/12 = 20n'\kappa$.
4. For any set of 3κ virtual servers, a randomly chosen set of $N/240 = n\kappa$ virtual servers in the first execution of Π_{GLOS} (resp. $N'/240 = n'\kappa$ virtual servers in the second execution of Π_{GLOS}) includes one of the 3κ virtual servers with an overwhelming probability.

For the first property, when $n > 2t$ in the first execution of Π_{GLOS}, the probability that each virtual server V_j is corrupted is $< (1/2)^6 = 1/64$. We set $X_j = 1$ if V_j is corrupted and $X_j = 0$ otherwise. We let $X = \sum_{j=1}^{N} X_j$ be the number of corrupted virtual servers. Since the random committees for emulating the virtual servers are selected independently, we have $\mathbb{E}(X) < N/64$. By Chernoff bound, we have:

$$P(X \geq N/60) < P(X \geq (1+1/15)\mathbb{E}(X)) \leq e^{-\frac{31N}{64\cdot 15}} = 2^{-\Omega(n\kappa)}.$$

Taking the union bound of all $< 2^n$ possible sets of corrupted parties, the probability is still negligible. Similarly, when $n' > 3t'$ in the second execution of Π_{GLOS}, the probability that each virtual server V_j is corrupted is $< (1/3)^6 < 1/120$, guaranteeing that the number of corrupted virtual servers is no more than $N'/120$ with an overwhelming probability. Therefore, the first property holds.

For the second property, for any set of $< t/2$ corrupted servers, the probability that each virtual server V_j is completely emulated by the remaining $s > n - t/2$ servers in this set but is not completely emulated by corrupted servers is at least

$$\left(\frac{s}{n}\right)^6 - \left(\frac{s-n/2}{n}\right)^6 \geq \left(\frac{n-t/2}{n}\right)^6 - \left(\frac{n/2-t/2}{n}\right)^6 > \left(\frac{3}{4}\right)^6 - \left(\frac{1}{4}\right)^6 > \frac{3}{17}.$$

We set $X_j = 1$ if V_j is honest and emulated completely by the remaining $s > n - t/2$ servers in this set, and $X_j = 0$ otherwise. We let $X = \sum_{j=1}^{N} X_j$ be the number of such virtual servers. Since the random committees for emulating the virtual servers are selected independently, we have $\mathbb{E}(X) > 3N/17$. By Chernoff bound, we have:

$$P(X \leq N/6) < P(X \leq (1 - 1/18)\mathbb{E}(X)) \leq e^{-\frac{3N}{18^3 \cdot 2}} = 2^{-\Omega(n\kappa)}.$$

Taking the union bound of all $\binom{n}{t} \cdot \binom{t}{s} < 3^n$ possible sets of all corrupted parties and the chosen $< n/2$ corrupted parties, the probability is still negligible. Therefore, the second property holds.

For the third property, the probability that each virtual server V_j is emulated completely by honest servers is $> (2/3)^6 = 2/23$. We set $X_j = 1$ if V_j is emulated completely by honest servers and $X_j = 0$ otherwise. We let $X = \sum_{j=1}^{N'} X_j$ be the number of such virtual servers. Since the random committees for emulating the virtual servers are selected independently, we have $\mathbb{E}(X) > 2N'/23$. By Chernoff bound, we have:

$$P(X \leq N'/12) < P(X \leq (1 - 1/24)\mathbb{E}(X)) \leq e^{-\frac{N'}{24^3}} = 2^{-\Omega(n'\kappa)}.$$

Taking the union bound of all $< 2^{n'}$ possible sets of corrupted parties, the probability is still negligible. Therefore, the third property holds.

For the last property, the probability that none of the 3κ virtual servers are included in this set is no more than:

$$\begin{aligned} &\frac{N - 3\kappa}{N} \cdot \frac{N - 1 - 3\kappa}{N - 1} \cdot \cdots \cdot \frac{N - N/240 + 1 - 3\kappa}{N - N/240 + 1} \\ &< \left(\frac{N - 3\kappa}{N}\right)^{n\kappa} < \left(1 - \frac{1}{80n}\right)^{n\kappa} < e^{\frac{\kappa}{80}} = 2^{-\Omega(\kappa)} \end{aligned}$$

in the first execution of Π_{GLOS}. Similar for the second one (only need to replace N with N'). Therefore, the last property also holds.

D The Protocol Description of Π_{share}

We give the detailed protocol description of Π_{share} here. We apply the same underlying protocol Π_0 as in [GLOS25], so we refer the readers to [GLOS25] for more details about it.

Protocol Π_{Share}

Let C be the circuit to be computed in the underlying protocol Π_0. The parties first run a public circuit preprocessing algorithm to transform C to a circuit C' with certain properties (see [GLOS25] for more details). Let G_A, W, W_O be the number of AND gates, wires, and output wires in C', respectively. The parties do the following:

1. **Input Sharings.** For each batch of k input wires attached to client $\mathtt{C}_i$, let $\boldsymbol{x} \in \mathbb{F}_2^k$ be the vector of the input values on the k wires. $\mathtt{C}_i$ generates a random $\Sigma^{(2)}$-sharing $[\boldsymbol{x}]^{(2)}$ based on the secret $\boldsymbol{x}$ and distributes it to the virtual servers, i.e. additively distributes each share $\boldsymbol{x}^{V_j}$ for virtual server V_j to $S_{j,1}, \ldots, S_{j,c}$.
2. **Mask Sharings for Outputs.** For each batch of k output wires attached to a client, the client samples a random $\Sigma^{(2)}$-sharing and distributes it to all the virtual servers.
3. **Random $\Sigma^{(2)}$-Sharings.** Each server S samples $10WN\ell/(n-t)k^2$ random $\Sigma^{(2)}$-sharings and distributes them to all the virtual servers. In addition, for each virtual server V_j, each member $V_{j,\alpha}$ (for $\alpha = 1, \ldots, c$) samples $10W/k^2$ random $\Sigma^{(2)}$-sharings and distributes them to all the virtual servers.
4. **Random Zero Sharings.** Each server S samples $10WN\ell/(n-t)k^2 + W_O/(n-t)k^2$ random $\Sigma^{(2)}$-sharings with all-zero secrets and distributes them to all the virtual servers.
5. **Sharings for the Output Labels.** For $i = 1, \ldots, \mathsf{rec}$ (where rec is the total number of reconstructions of $\Sigma^{(2)}$-sharings in the underlying protocol), if the receiver R_i of the i-th reconstruction in the evaluation phase of the underlying protocol 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 $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}$ and a random Σ-sharing as $[\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$. Then $S_{j,\beta}$ distributes each pair of $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ to the virtual servers. If R_i is a client $\mathtt{C}_j$, he plays as c servers of a virtual server to generate and distribute the sharings.

Figure 10: The sharing phase for the multiparty garbling.

E The Functionality $\mathcal{F}_{\mathsf{GLOSBaseROT}}$

The functionality $\mathcal{F}_{\mathsf{GLOSBaseROT}}$ is provided as follows.

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

Upon receiving ROT from all the honest servers, for each pair of members $(S_{j,\alpha}, S_{j,\beta})$ of each virtual server V_j, the trusted party does the following κ times:

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

Figure 11: Functionality for base random oblivious transfers in Π_{GLOS}.

F The Detailed Garbling Protocol for Virtual Servers

We introduce the detailed garbling protocol $\Pi_{\mathsf{Gb}}(V_j)$ for each virtual server V_j here.

F.1 Subprotocols for the Garbling

We first present basic subprotocols for the garbling. The first one Π_{Open} is to let c parties open the secret of an additive sharing.

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 12: Protocol for opening the secret of an additive sharing.

The second subprotocol is the multiplication protocol Π_{Mult} for two additive shared values. The multiplication process is instantiated with pair-wise ROTs.

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 $\langle \boldsymbol{z} \rangle = \langle \boldsymbol{x} \cdot y \rangle$ ($\boldsymbol{x} \in \mathbb{F}_2^\kappa, y \in \mathbb{F}_2$) 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 13: Protocol for multiplication.

F.2 Garbling Protocol Description

Now we are ready to present the garbling protocol. The parties first run a local computation phase to prepare the preprocessing data for the underlying protocol and the output labels for the garbled circuit. Then, the parties run a standard semi-honest BMR-type garbling. The protocol $\Pi_{\mathsf{Gb}}(V_j)$ for garbling V_j's computation $\mathtt{Circ}^{V_j}$ in the underlying protocol Π_0 in [GLOS25] (we refer the readers to [GLOS25] for more details about the underlying protocol Π_0) is given as follows.

Protocol $\Pi_{\mathsf{Gb}}(V_j)$

Let C be the circuit to be computed in Π_0. The protocol runs between c servers $S_{j,1}, \ldots, S_{j,c}$, who agree on a circuit $\mathtt{Circ}^{V_j}$ to be garbled (i.e. V_j's local computation in evaluation phase of Π_0). $\mathtt{Circ}^{V_j}$ consists of AND, XOR, and output gates.

Inputs: The inputs to $\Pi_{\mathsf{Gb}}(V_j)$ contain the following:

1. The servers input all the shares they received during the execution of Π_{Share}.
2. Each server $S_{j,i}$ inputs a κ-bit random string as the randomness for committing to the other inputs.
3. Each server $S_{j,i}$ inputs a κ-bit seed used for generating his randomness (via RO) during the garbling, including:
   (a) A wire label $k_{w,0}^{S_{j,i}} \in \mathbb{F}_2^\kappa$ and a share $\lambda_w^{S_{j,i}}$ of the additive sharing $\langle \lambda_w \rangle$ of wire mask $\lambda_w \in \{0,1\}$ for each wire of $\mathtt{Circ}^{V_j}$ except output wires of XOR and output gates.
   (b) A global κ-bit string $\Delta^{S_{j,i}}$ as the sum (over $\mathbb{F}_2$) of each his two wire labels of each wire.
   (c) His randomness for the OT extension in [KOS15] executed with each $S_{j,\alpha}, \alpha \neq i$.
4. Each pair of servers input κ instances of ROTs.

The protocol proceeds as follows:

1. **Local Computation.** The servers first perform a local computation process to prepare the additive sharing for all the preprocessing data for V_j in Π_0 (see [GLOS25] for more details). Then, the servers locally compute the output labels for the garbling of $\mathtt{Circ}^{V_j}$ by:
   (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

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

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

   where $([\boldsymbol{s}])^{V_j}$ denotes V_j's share of $[\boldsymbol{s}]$ and $([\boldsymbol{s}]^{(3)})_{[c_1,c_2]}^{V_j}$ 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 $Y_{\mathsf{k},b}^{S_{j,\beta}} = (\langle Y_{\mathsf{k},b,1}^{V_j} \rangle^{S_{j,\beta}}, \ldots, \langle Y_{\mathsf{k},b,c}^{V_j} \rangle^{S_{j,\beta}})$ 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. **OT Extension.** Each pair of servers invokes the OT extension protocol Π_{OText} in [KOS15] (the fixed version based on the techniques in [Roy22]) to get $(4c+1)\cdot G_A^{V_j} + 2c\ell W_O^{V_j}$ instances of ROTs with message length κ. Here $G_A^{V_j}$ is the number of AND gates in $\mathtt{Circ}^{V_j}$ and $W_O^{V_j}$ is the number of output wires in $\mathtt{Circ}^{V_j}$.

3. **Handling XOR gates.** For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, each server $S_{j,i}$ computes
$$k_{o,0}^{S_{j,i}} \| \lambda_o^{S_{j,i}} = (k_{a,0}^{S_{j,i}} \| \lambda_a^{S_{j,i}}) \oplus (k_{b,0}^{S_{j,i}} \| \lambda_b^{S_{j,i}}).$$
This computation is performed gate by gate.

4. **Computing 1-Labels.** 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 $k_{w,1}^{S_{j,i}} = k_{w,0}^{S_{j,i}} \oplus \Delta^{S_{j,i}}$.

5. **Handling AND Gates.** For each AND gate g in $\mathtt{Circ}^{V_j}$ with input wires a, b and output wire o, the servers hold $\langle k_{o,0}^{S_{j,\alpha}} \rangle$ where $S_{j,\alpha}$'s share is $k_{o,0}^{S_{j,\alpha}}$ and all other servers have all-0 shares. Similarly, they hold $\langle k_{o,1}^{S_{j,\alpha}} \rangle$, and they also hold $\langle \lambda_a \rangle, \langle \lambda_b \rangle, \langle \lambda_o \rangle$. Then:
   (a) Each pair of servers takes 1 instance of ROT. 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
$$\chi_1 = ((0 \oplus \lambda_a) \wedge (0 \oplus \lambda_b)) \oplus \lambda_o, \qquad \chi_2 = ((0 \oplus \lambda_a) \wedge (1 \oplus \lambda_b)) \oplus \lambda_o,$$
$$\chi_3 = ((1 \oplus \lambda_a) \wedge (0 \oplus \lambda_b)) \oplus \lambda_o, \qquad \chi_4 = ((1 \oplus \lambda_a) \wedge (1 \oplus \lambda_b)) \oplus \lambda_o.$$
   (b) Each pair of servers takes $4c$ instances of ROTs. Then the servers run Π_{Mult} to compute $\langle \chi_i \cdot (k_{o,1}^{S_{j,\alpha}} - k_{o,0}^{S_{j,\alpha}}) \rangle$ for $i = 1, 2, 3, 4$ and $\alpha = 1, \ldots, c$. Then the servers locally compute $\langle k_{o,\chi_i}^{S_{j,\alpha}} \rangle = \langle k_{o,0}^{S_{j,\alpha}} \oplus \chi_i \cdot (k_{o,1}^{S_{j,\alpha}} - k_{o,0}^{S_{j,\alpha}}) \rangle$ for $i = 1, 2, 3, 4$ and $\alpha = 1, \ldots, c$.
   (c) Each server $S_{j,i}$ calls the random oracle $\mathcal{O}$ with input $k_{a,i_0}^{S_{j,i}} \| i_0 \| k_{b,i_1}^{S_{j,i}} \| i_1 \| i \| j \| \alpha \| g$ for each $(i_0, i_1) = (0,0), (0,1), (1,0), (1,1)$ and $\alpha = 1, \ldots, c$ 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.
   (d) The servers locally compute $\langle A_{g,2i_0+i_1}^{S_{j,\alpha}} \rangle$ for each $(i_0, i_1) = (0,0), (0,1), (1,0), (1,1)$ and $\alpha = 1, \ldots, c$, where
$$A_{g,2i_0+i_1}^{S_{j,\alpha}} = \left(\bigoplus_{i=1}^{c} \left(\mathcal{O}(k_{a,i_0}^{S_{j,i}} \| i_0 \| k_{b,i_1}^{S_{j,i}} \| i_1 \| i \| j \| \alpha \| g) \right) \right) \oplus (k_{o,\chi_{2i_0+i_1}}^{S_{j,\alpha}} \| \chi_{2i_0+i_1}),$$
   Let $\boldsymbol{A}_{g,1}^{V_j} = (A_{g,1}^{S_{j,1}}, \ldots, A_{g,1}^{S_{j,c}})$ and similar for $\boldsymbol{A}_{g,2}^{V_j}, \boldsymbol{A}_{g,3}^{V_j}, \boldsymbol{A}_{g,4}^{V_j}$. The servers then get $\langle \boldsymbol{A}_{g,1}^{V_j} \rangle, \langle \boldsymbol{A}_{g,2}^{V_j} \rangle, \langle \boldsymbol{A}_{g,3}^{V_j} \rangle, \langle \boldsymbol{A}_{g,4}^{V_j} \rangle$.

6. **Handling Output Gates.** For each output gate with index k in $\mathtt{Circ}^{V_j}$ with input wire w, the servers hold $\langle Y_{\mathsf{k},0}^{S_{j,\alpha}} \rangle$ where $S_{j,\alpha}$'s share is $Y_{\mathsf{k},0}^{S_{j,\alpha}}$ and all other servers have all-0 shares. Similarly, they hold $\langle Y_{\mathsf{k},1}^{S_{j,\alpha}} \rangle$, and they also hold $\langle \lambda_w \rangle$. Then:
   (a) Each pair of servers takes $2c\ell$ instances of ROTs. Then the servers run Π_{Mult} to compute $\langle (i_2 \oplus \lambda_w) \cdot (Y_{\mathsf{k},1}^{S_{j,\alpha}} - Y_{\mathsf{k},0}^{S_{j,\alpha}}) \rangle$ for $i_2 = 0, 1$ and $\alpha = 1, \ldots, c$. Then the servers locally compute $\langle Y_{\mathsf{k},i_2 \oplus \lambda_w}^{S_{j,\alpha}} \rangle = \langle Y_{\mathsf{k},0}^{S_{j,\alpha}} \oplus (i_2 \oplus \lambda_w) \cdot (Y_{\mathsf{k},1}^{S_{j,\alpha}} - Y_{\mathsf{k},0}^{S_{j,\alpha}}) \rangle$ for $i_2 = 0, 1$ and $\alpha = 1, \ldots, c$.
   (b) Each server $S_{j,i}$ calls the random oracle $\mathcal{O}$ with input $k_{w,i_2}^{S_{j,\alpha}} \| i_2 \| i \| j \| \alpha \| w$ 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 $S_{j,1}, \ldots, S_{j,3}$, where all the servers except $S_{j,i}$ have all-0 shares.
   (c) The servers locally compute $\langle \mathsf{ct}_{w,i_2}^{S_{j,\alpha}} \rangle$ for each $i_2 = 0, 1$ and $\alpha = 1, \ldots, c$, where
$$\mathsf{ct}_{w,i_2}^{S_{j,\alpha}} = \left(\bigoplus_{i=1}^{3} \left(\mathcal{O}(k_{w,i_2}^{S_{j,i}} \| i_2 \| i \| j \| \alpha \| w) \right) \right) \oplus Y_{\mathsf{k},i_2 \oplus \lambda_w}^{S_{j,\alpha}},$$
   Let $\mathbf{ct}_{w,i_2}^{V_j} = (\mathsf{ct}_{w,i_2}^{S_{j,1}}, \ldots, \mathsf{ct}_{w,i_2}^{S_{j,c}})$ for each $i_2 = 0, 1$. The servers then get $\langle \mathbf{ct}_{w,0}^{V_j} \rangle, \langle \mathbf{ct}_{w,1}^{V_j} \rangle$.

7. **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 $\Pi_{\mathsf{Open}}(\langle x_w \oplus \lambda_w \rangle)$. Then, the servers output their shares of $\langle x_w \oplus \lambda_w \rangle$.
8. **Outputting Output Masks.** For each input wire w of an output gate in $\mathtt{Circ}^{V_j}$, the servers output their shares of $\langle \lambda_w \rangle$.
9. **Encrypting Input Labels.** For each reconstruction of $\Sigma^{(2)}$-sharing in the evaluation phase of Π_0 whose receiver is V_j, we assume it's the i-th reconstruction, and the sharing to be reconstructed is $[\boldsymbol{s}_i]^{(2)}$. Suppose the η-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_η in $\mathtt{Circ}^{V_j}$:
   (a) Each server $S_{j,\beta}$ queries the random oracle $\mathcal{O}$ with inputs $\boldsymbol{r}_{0,\eta,\beta} \| 0 \| i \| \beta \| \eta \| j_\eta$ and $\boldsymbol{r}_{1,\eta,\beta} \| 1 \| i \| \beta \| \eta \| j_\eta$, where $\boldsymbol{r}_{b,\eta,\beta} = (r^{(1)}_{b,\eta,\beta}, \ldots, r^{(\kappa)}_{b,\eta,\beta})$ for each $b = 0, 1$ with each $\boldsymbol{r}^{(\alpha)}_{b,\beta} = (r^{(\alpha)}_{b,1,\beta}, \ldots, r^{(\alpha)}_{b,k,\beta})$. Then $S_{j,\beta}$ receives $\mathcal{O}(\boldsymbol{r}_{0,\eta,\beta} \| 0 \| i \| \beta \| \eta \| j_\eta)$ and $\mathcal{O}(\boldsymbol{r}_{1,\eta,\beta} \| 1 \| i \| \beta \| \eta \| j_\eta)$.
   (b) Each server $S_{j,\beta}$ locally computes
   $$\mathsf{ct}^{(i,\beta)}_{j_\eta,0} = \mathcal{O}(\boldsymbol{r}_{0,\eta,\beta} \| 0 \| i \| \beta \| \eta \| j_\eta) \oplus \left(k^{S_{j,\beta}}_{w_{j_\eta},\lambda_{w_{j_\eta}}} \| \lambda_{w_{j_\eta}}\right)$$
   and
   $$\mathsf{ct}^{(i,\beta)}_{j_\eta,1} = \mathcal{O}(\boldsymbol{r}_{1,\eta,\beta} \| 1 \| i \| \beta \| \eta \| j_\eta) \oplus \left(k^{S_{j,\beta}}_{w_{j_\eta},1\oplus\lambda_{w_{j_\eta}}} \| (1 \oplus \lambda_{w_{j_\eta}})\right).$$
   Then $S_{j,\beta}$ outputs the ciphertexts $\mathsf{ct}^{(i,\beta)}_{j_\eta,0}, \mathsf{ct}^{(i,\beta)}_{j_\eta,1}$.
10. **Outputting Input Labels.** For each input wire w of $\mathtt{Circ}^{V_j}$, if the wire value x_w doesn't come from a reconstruction of a $\Sigma^{(2)}$-sharing in the evaluation phase of Π_0:
    (a) The servers $S_{j,1}, \ldots, S_{j,c}$ output $x_w \oplus \lambda_w$.
    (b) Each server $S_{j,i}$ outputs $k^{S_{j,i}}_{w,x_w\oplus\lambda_w}$.
    The above outputs can be viewed as an additive sharing of the input label $k^{S_{j,1}}_{w,x_w\oplus\lambda_w} \| \ldots \| k^{S_{j,c}}_{w,x_w\oplus\lambda_w} \| (x_w \oplus \lambda_w)$ on wire w.
11. **Outputting Garbled Circuits.** Each server $S_{j,\beta}$ outputs 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.

Figure 14: The garbling process of [GLOS25] for each virtual server.

In particular, let each server $S_{j,i}$'s input to $\Pi_{\mathsf{Gb}}(V_j)$ be $I_{j,i}$. In the garbling phase, $S_{j,i}$ first commits to his inputs by querying the random oracle with $I_{j,i} \| \sigma_{j,i}$ for a random κ-bit string $\sigma_{j,i}$ and sending $\mathcal{O}(I_{j,i} \| \sigma_{j,i})$ to all the servers. Then, the parties $S_{j,1}, \ldots, S_{j,c}$ jointly run $\Pi_{\mathsf{Gb}}(V_j)$.

G Description and Security Proof of $\mathsf{Sim}_{\mathsf{GLOS}}$

In this section, we provide the formal description and security proof of $\mathsf{Sim}_{\mathsf{GLOS}}$. Let $\mathcal{T}_{\mathsf{cor}}$ be the set of corrupted virtual servers. We recall our requirements on $\mathsf{Sim}_{\mathsf{GLOS}}$ introduced in Section 4.3. Concretely speaking, $\mathsf{Sim}_{\mathsf{GLOS}}$ first receives a set $\mathcal{T}_{\mathsf{ver}}$ containing at most $T - |\mathcal{T}_{\mathsf{cor}}|$ honest virtual servers. We require $\mathsf{Sim}_{\mathsf{GLOS}}$ to simulate the corrupted parties' views during the protocol in [GLOS25] before the verification phase. In addition, for each honest virtual server $V_j \in \mathcal{T}_{\mathsf{ver}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ simulates the views of honest servers in $\Pi_{\mathsf{Gb}}(V_j)$. Then $\mathsf{Sim}_{\mathsf{GLOS}}$ receives sets $\mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}$, where $\mathcal{T}_{\mathsf{hon}}$ is a subset of $\mathcal{T}_{\mathsf{act}}$. It holds that:

- $\mathcal{T}_{\mathsf{hon}} \cap (\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}}) = \emptyset$;
- $|\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}| - |\mathcal{T}_{\mathsf{hon}}| \leq T$.
- For each $V_j \in \mathcal{T}_{\mathsf{hon}}$, servers honestly emulate V_j:
  - Each corrupted server among $S_{j,1}, \ldots, S_{j,c}$ runs $\Pi_{\mathsf{Gb}}(V_j)$ honestly from his committed input (Recall that these inputs are committed via the random oracle before the servers run $\Pi_{\mathsf{Gb}}(V_j)$).

– The shares generated in Π_{Share} for the virtual servers in $\mathcal{T}_{\mathsf{hon}}$ are all from valid sharings. More concretely:

  * For each sharing generated by an honest dealer in Π_{Share}, the shares sent to each corrupted server $S_{j,\alpha}$ (where $V_j \in \mathcal{T}_{\mathsf{hon}}$) match $S_{j,\alpha}$'s committed input to $\Pi_{\mathsf{Gb}}(V_j)$.
  * For each Σ-sharing generated by a corrupted dealer in Π_{Share}, there exists a valid Σ-sharing $[\boldsymbol{v}]$ such that for each $V_j \in \mathcal{T}_{\mathsf{act}}$, the shares of $\langle([\boldsymbol{v}])^{V_j}\rangle$ (where $([\boldsymbol{v}])^{V_j}$ is V_j's share of $[\boldsymbol{v}]$) for the honest servers among $S_{j,1}, \ldots, S_{j,c}$ match the messages sent by the corrupted dealer in Π_{Share}, and the shares for corrupted servers match their committed inputs to $\Pi_{\mathsf{Gb}}(V_j)$. Similar for the $\Sigma^{(2)}, \Sigma^{(3)}$-sharings and the $\Sigma^{(2)}$-sharings with all-zero secrets.

– The input base ROTs (to $\Pi_{\mathsf{Gb}}(V_j)$) between each pair of servers $(S_{j,\alpha}, S_{j,\beta})$ emulating a virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ match what they received from $\mathcal{F}_{\mathsf{GLOSBaseROT}}$.

For each virtual server V_j in $\mathcal{T}_{\mathsf{act}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ outputs the honest servers' outputs from $\Pi_{\mathsf{Gb}}(V_j)$.

To formalize such a simulator, we follow the real and ideal world paradigm [Can00]. Briefly speaking, we construct a functionality $\mathcal{F}_{\mathsf{GLOS}}$ and a specific modified protocol Π from [GLOS25], where Π securely realizes $\mathcal{F}_{\mathsf{GLOS}}$, and $\mathsf{Sim}_{\mathsf{GLOS}}$ acts as the simulator in the security proof of Π.

G.1 The Ideal World

In the ideal world, the simulator $\mathsf{Sim}_{\mathsf{GLOS}}$ interacts with a functionality $\mathcal{F}_{\mathsf{GLOS}}$. The inputs of the parties to $\mathcal{F}_{\mathsf{GLOS}}$ not only contain the function inputs of the clients but also the sets $\mathcal{T}_{\mathsf{ver}}, \mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}$ from honest servers. In particular, $\mathcal{F}_{\mathsf{GLOS}}$ first delivers $\mathcal{T}_{\mathsf{ver}}$ and then delivers $\mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}$ to $\mathsf{Sim}_{\mathsf{GLOS}}$.

$\mathcal{F}_{\mathsf{GLOS}}$ acts as a standard MPC functionality to receive inputs and send corrupted clients' function outputs to the adversary. However, honest clients do not have outputs, and we let the ideal adversary decide honest servers' outputs. In this way, we can later set the honest servers' outputs in the real world to their views during $\Pi_{\mathsf{Gb}}(V_j)$ for each $V_j \in \mathcal{T}_{\mathsf{ver}}$ and their outputs from $\Pi_{\mathsf{Gb}}(V_j)$ for each $V_j \in \mathcal{T}_{\mathsf{act}}$, and then the simulator $\mathsf{Sim}_{\mathsf{GLOS}}$ needs to simulate them as desired.

The functionality $\mathcal{F}_{\mathsf{GLOS}}$ is presented below.

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

1. The trusted party receives $\mathcal{T}_{\mathsf{ver}}$ from honest servers and forwards it to $\mathsf{Sim}_{\mathsf{GLOS}}$.
2. The trusted party receives $\mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}$ from honest servers and forwards them to $\mathsf{Sim}_{\mathsf{GLOS}}$.
3. Upon receiving the inputs from all the clients, the trusted party computes each corrupted client's output of circuit C, and sends it to $\mathsf{Sim}_{\mathsf{GLOS}}$.
4. Honest clients do not have outputs. The trusted party receives honest servers' outputs from $\mathsf{Sim}_{\mathsf{GLOS}}$ and sends them to the honest servers.

Figure 15: Functionality $\mathcal{F}_{\mathsf{GLOS}}$.

We note that $\mathcal{F}_{\mathsf{GLOS}}$ does not put any correctness requirement on honest servers' outputs. As a result, an empty protocol (where parties do not do anything) trivially realizes $\mathcal{F}_{\mathsf{GLOS}}$. However, in this section, $\mathcal{F}_{\mathsf{GLOS}}$ is only used to model the properties of $\mathsf{Sim}_{\mathsf{GLOS}}$. Our construction does not use $\mathcal{F}_{\mathsf{GLOS}}$ directly. Instead, the simulator of our main protocol invokes $\mathsf{Sim}_{\mathsf{GLOS}}$ and emulates $\mathcal{F}_{\mathsf{GLOS}}$ to simulate the required information. Our construction also relies on other properties of the protocol in [GLOS25], so other protocols that achieve the same functionality cannot be used directly in our construction.

G.2 The Real World

Next, we present the protocol Π modified from the protocol in [GLOS25]. Ideally, the environment forwards the function inputs to all clients and the set $\mathcal{T}_{\mathsf{ver}}$ to all servers. Then, all parties run the protocol in [GLOS25] before the verification phase. After that, all servers receive $\mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}$ from the environment. Finally, for each

virtual server V_j in $\mathcal{T}_{\mathsf{hon}}$, servers emulating V_j output their output of $\Pi_{\mathsf{Gb}}(V_j)$. For each virtual server V_j in $\mathcal{T}_{\mathsf{ver}}$, servers emulating V_j output their views in $\Pi_{\mathsf{Gb}}(V_j)$.

However, a subtle issue is that $\mathsf{Sim}_{\mathsf{GLOS}}$ can only simulate honest servers' outputs when $\mathcal{T}_{\mathsf{ver}}, \mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}$ satisfy certain requirements (as we mentioned at the beginning of this section), and these requirements cannot be verified by honest servers. To address this issue, we introduce a functionality $\mathcal{F}_{\mathsf{emulate}}$ to help verify these sets. In particular, $\mathcal{F}_{\mathsf{emulate}}$ serves as the P2P channel across the protocol execution and emulates the random oracle as well as the ideal functionalities $\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{GLOSbaseROT}}$ in the protocol. In this way, $\mathcal{F}_{\mathsf{emulate}}$ can completely monitor corrupted servers' behaviors and can verify the sets. Honest servers will only output their views or outputs of $\Pi_{\mathsf{Gb}}(V_j)$ as required when $\mathcal{F}_{\mathsf{emulate}}$ returns $\mathsf{success}$, and $\perp$ otherwise.

The functionality $\mathcal{F}_{\mathsf{emulate}}$ is presented below.

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

1. Upon receiving $(\mathsf{send}, P, P', m)$ from party P, the trusted party sends (P, m) to P'.
2. Upon receiving (query, P, q) from party P, the trusted party faithfully emulates the random oracle $\mathcal{O}$ on input q to send $\mathcal{O}(q)$ to P.
3. Upon receiving $\mathsf{RandCoin}$ from all the honest servers, the trusted party faithfully emulates $\mathcal{F}_{\mathsf{Coin}}$ to send the outputs to servers.
4. Upon receiving ROT from all the honest servers, the trusted party faithfully emulates $\mathcal{F}_{\mathsf{GLOSbaseROT}}$ to send the outputs to servers.
5. Upon receiving $(\mathsf{ver}, \mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}, \mathcal{T}_{\mathsf{ver}})$ from all the parties, where $\mathcal{T}_{\mathsf{hon}}$ is a subset of $\mathcal{T}_{\mathsf{act}}$, the trusted party verifies whether the following hold:
   - Let $\mathcal{T}_{\mathsf{cor}}$ be the set of virtual servers that are completely emulated by corrupted servers. It holds that $\mathcal{T}_{\mathsf{hon}} \cap (\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}}) = \emptyset$ and $|\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}| - |\mathcal{T}_{\mathsf{hon}}| \leq T$.
   - For each $V_j \in \mathcal{T}_{\mathsf{hon}}$, corrupted servers follow the protocol to run $\Pi_{\mathsf{Gb}}(V_j)$ with their committed inputs and randomness (before executing $\Pi_{\mathsf{Gb}}(V_j)$). In addition, the committed correlated randomness matches that received from $\mathcal{F}_{\mathsf{emulate}}$.
   - For each $V_j \in \mathcal{T}_{\mathsf{hon}}$ and each Σ-sharing generated by a corrupted dealer in Π_{Share}, there exists a valid Σ-sharing $[\boldsymbol{v}]$ such that for each $V_j \in \mathcal{T}_{\mathsf{act}}$, the shares of $\langle([\boldsymbol{v}])^{V_j}\rangle$ (where $([\boldsymbol{v}])^{V_j}$ is V_j's share of $[\boldsymbol{v}]$) for the honest servers among $S_{j,1}, \ldots, S_{j,c}$ match the messages sent by the corrupted dealer in Π_{Share}, and the shares for corrupted servers match their inputs to $\Pi_{\mathsf{Gb}}(V_j)$. Similar for the $\Sigma^{(2)}, \Sigma^{(3)}$-sharings and the $\Sigma^{(2)}$-sharings with all-zero secrets.

   If true, the trusted party sends $\mathsf{success}$ to all the servers.

Figure 16: Functionality $\mathcal{F}_{\mathsf{emulate}}$.

Jumping ahead, these views of corrupted servers are fully committed in our main protocol, where the simulator for our main protocol knows them. Therefore, the simulator for our main protocol is able to emulate the corrupted parties and run $\mathsf{Sim}_{\mathsf{GLOS}}$ with these views to receive the required output from $\mathsf{Sim}_{\mathsf{GLOS}}$. Thus, introducing $\mathcal{F}_{\mathsf{emulate}}$ in the protocol will not lead to additional difficulties in proving the security of our main protocol.

After running the protocol of [GLOS25] before the verification, the honest servers will send the sets $\mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{ver}}, \mathcal{T}_{\mathsf{hon}}$ to $\mathcal{F}_{\mathsf{emulate}}$. If $\mathsf{success}$ is received, showing that our requirements are satisfied, each server $S_{j,\alpha}$ emulating a virtual server $V_j \in \mathcal{T}_{\mathsf{act}}$ will output his outputs from $\Pi_{\mathsf{Gb}}(V_j)$. Similarly, each server $S_{j,\alpha}$ emulating a virtual server $V_j \in \mathcal{T}_{\mathsf{ver}}$ will output his view during $\Pi_{\mathsf{Gb}}(V_j)$. In this case, $\mathsf{Sim}_{\mathsf{GLOS}}$ should simulate these outputs as desired when proving the security of the protocol Π. Otherwise, the sets do not satisfy our requirements, so we do not require $\mathsf{Sim}_{\mathsf{GLOS}}$ to simulate anything else. In this case, we can simply let the honest servers output $\perp$.

The protocol Π is provided below.

Protocol Π

1. The clients receive their function inputs and the servers receive the set $\mathcal{T}_{\mathsf{ver}}$ from the environment.
2. The parties (including clients and servers) call $\mathcal{F}_{\mathsf{emulate}}$ with $\mathsf{RandCoin}$ to obtain a κ-bit string r_{vp} that determines the committee of parties emulating each of the N virtual servers.
3. The parties jointly run Π_{Share} via $\mathcal{F}_{\mathsf{emulate}}$. More concretely, when each party P needs to send a message m to another party P' in Π_{Share}, he sends $(\mathsf{send}, P, P', m)$ to $\mathcal{F}_{\mathsf{emulate}}$.
4. The servers send ROT to $\mathcal{F}_{\mathsf{emulate}}$ to prepare random OT correlations.
5. Let each server $S_{j,i}$'s input to $\Pi_{\mathsf{Gb}}(V_j)$ form a set $I_{j,i}$. $S_{j,i}$ commits to his inputs by sampling a random κ-bit string $\sigma_{j,i}$, sending $(\mathsf{query}, S_{j,i}, I_{j,i} \| \sigma_{j,i})$ to $\mathcal{F}_{\mathsf{emulate}}$, and forwarding the received output $\mathcal{O}(I_{j,i} \| \sigma_{j,i})$ to all the servers.
6. For each $j = 1, \ldots, N$, the servers $S_{j,1}, \ldots, S_{j,\alpha}$ run $\Pi_{\mathsf{Gb}}(V_j)$ via $\mathcal{F}_{\mathsf{emulate}}$ with their received preprocessing data and input shares. More concretely, when each party P needs to send a message m to another party P' in Π_{Share}, he sends $(\mathsf{send}, P, P', m)$ to $\mathcal{F}_{\mathsf{emulate}}$. When each party P needs to query the random oracle with a query q, he sends (query, P, q) to $\mathcal{F}_{\mathsf{emulate}}$.
7. Then servers receive the sets $\mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}$ from the environment and send $(\mathsf{ver}, \mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}, \mathcal{T}_{\mathsf{ver}})$ to $\mathcal{F}_{\mathsf{emulate}}$.
8. If $\mathsf{success}$ is received from $\mathcal{F}_{\mathsf{emulate}}$ in the last step, for each virtual server $V_j \in \mathcal{T}_{\mathsf{act}}$, each server $S_{j,\alpha}$ emulating V_j outputs his output from $\Pi_{\mathsf{Gb}}(V_j)$. Then, for each virtual server $V_j \in \mathcal{T}_{\mathsf{ver}}$, each server $S_{j,\alpha}$ emulating V_j outputs his view during the execution of $\Pi_{\mathsf{Gb}}(V_j)$. Otherwise, the servers output $\perp$.

Figure 17: Protocol Π.

G.3 Description of $\mathsf{Sim}_{\mathsf{GLOS}}$

In this section, we provide a detailed description of $\mathsf{Sim}_{\mathsf{GLOS}}$. Recall that in each garbling protocol $\Pi_{\mathsf{Gb}}(V_j)$, we need each pair of servers to run the OT extension protocol Π_{OText}. In the construction of $\mathsf{Sim}_{\mathsf{GLOS}}$, we need to invoke the simulator in the security proof of Π_{OText}, which relies on the fact that Π_{OText} (parametrized by the number L of ROT correlations required for each pair of servers in $\Pi_{\mathsf{Gb}}(V_j)$) UC-securely computes the two-party functionality $\mathcal{F}_{\mathsf{ROT}}(L)$ (Figure 18) below in the $\mathcal{F}_{\mathsf{ROT}}(\kappa)$-hybrid model, where $\mathcal{F}_{\mathsf{ROT}}(\kappa)$ is only invoked once at the beginning. The simulator for proving the security of Π_{OText} is denoted by $\mathsf{Sim}_{\mathsf{OText}}$.

Functionality $\mathcal{F}_{\mathsf{ROT}}(L)$

Upon receiving (ROT, L) from the parties A, B, the trusted party does the following L times:

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

Figure 18: Functionality for random oblivious transfers after OT extension between two parties A, B.

The detailed description of $\mathsf{Sim}_{\mathsf{GLOS}}$ is presented below in Figure 19.

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

To simulate the environment $\mathcal{Z}$, Sim forwards every message it receives from the environment $\mathcal{Z}$ to $\mathcal{A}$ and vice-versa.

1. For each $j = 1, \ldots, N$, $\mathsf{Sim}_{\mathsf{GLOS}}$ sets $\mathsf{Comp}_j = 0$.
2. $\mathsf{Sim}_{\mathsf{GLOS}}$ receives $\mathcal{T}_{\mathsf{ver}}$ from $\mathcal{F}_{\mathsf{GLOS}}$.
3. $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive $\mathsf{RandCoin}$ from all the corrupted parties and follows the protocol to sample r_{vp} randomly. Then $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to send r_{vp} to $\mathcal{A}$. If abort is received from $\mathcal{A}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{Coin}}$ to send abort to all the corrupted parties and aborts the protocol. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ follows the protocol to determine the virtual servers and receive the result of $\{S_{j,1}, \ldots, S_{j,c}\}$ for each $j = 1, \ldots, N$.

4. For each honest virtual server $V_j \notin \mathcal{T}_{\mathsf{ver}}$:
   - Let $L = (4c+1) \cdot G_A^{V_j} + 2c\ell W_O^{V_j}$. For each invocation of $\mathcal{F}_{\mathsf{GLOSbaseROT}}$ between each pair of servers $(S_{j,\alpha}, S_{j,\beta})$ where $S_{j,\alpha}$ is honest and $S_{j,\beta}$ is corrupted, $\mathsf{Sim}_{\mathsf{GLOS}}$ receives $(b^{(\alpha,\beta)}, r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)})$ from $S_{j,\beta}$. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ initializes an adversary $\mathcal{A}_{\mathsf{OText}}$ for Π_{OText} (running between $S_{j,\alpha}, S_{j,\beta}$) that sends the same messages to $\mathcal{F}_{\mathsf{ROT}}(\kappa)$ and runs $\mathsf{Sim}_{\mathsf{OText}}$ with this adversary. $\mathsf{Sim}_{\mathsf{GLOS}}$ then receives the corresponding output for $S_{j,\beta}$ from $\mathcal{F}_{\mathsf{ROT}}(\kappa)$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send it to $S_{j,\beta}$. Whenever $\mathsf{Sim}_{\mathsf{OText}}$ invokes $\mathcal{F}_{\mathsf{ROT}}(L)$, $\mathsf{Sim}_{\mathsf{GLOS}}$ faithfully emulates $\mathcal{F}_{\mathsf{ROT}}(L)$ to receive $S_{j,\beta}$'s output of $\mathcal{F}_{\mathsf{ROT}}(L)$ and send it to $\mathsf{Sim}_{\mathsf{GLOS}}$.
   - Similarly, for each invocation of $\mathcal{F}_{\mathsf{emulate}}$ of each pair of servers $(S_{j,\alpha}, S_{j,\beta})$ where $S_{j,\beta}$ is honest and $S_{j,\alpha}$ is corrupted, $\mathsf{Sim}_{\mathsf{GLOS}}$ receives $(r_0^{(\alpha,\beta)}, r_1^{(\alpha,\beta)})$ from $S_{j,\alpha}$. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ initializes an adversary $\mathcal{A}_{\mathsf{OText}}$ for Π_{OText} (running between $S_{j,\alpha}, S_{j,\beta}$) that sends the same messages to $\mathcal{F}_{\mathsf{ROT}}(\kappa)$ and runs $\mathsf{Sim}_{\mathsf{OText}}$ with this adversary. $\mathsf{Sim}_{\mathsf{GLOS}}$ then receives the corresponding output for $S_{j,\alpha}$ from $\mathcal{F}_{\mathsf{ROT}}(\kappa)$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send it to $S_{j,\alpha}$. Whenever $\mathsf{Sim}_{\mathsf{OText}}$ invokes $\mathcal{F}_{\mathsf{ROT}}(L)$, $\mathsf{Sim}_{\mathsf{GLOS}}$ faithfully emulates $\mathcal{F}_{\mathsf{ROT}}(L)$ to receive $S_{j,\alpha}$'s output of $\mathcal{F}_{\mathsf{ROT}}(L)$ and send it to $\mathsf{Sim}_{\mathsf{GLOS}}$.

   For other invocations of $\mathcal{F}_{\mathsf{GLOSBaseROT}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ faithfully emulates $\mathcal{F}_{\mathsf{emulate}}$.
5. $\mathsf{Sim}_{\mathsf{GLOS}}$ runs $\mathsf{Sim}_{\mathsf{GLOSShare}}$ (Figure 20).
6. $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive corrupted parties' inputs to $\Pi_{\mathsf{Gb}}(V_j)$ for each virtual server V_j. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ checks whether the following holds:
   - For each corrupted server $S_{j,\alpha}$'s share generated by honest parties during Π_{Share}, the committed share matches what $\mathsf{Sim}_{\mathsf{GLOS}}$ sends to $S_{j,\alpha}$.
   - For each ROT instance, the committed outputs from $\mathcal{F}_{\mathsf{GLOSbaseROT}}$ of corrupted parties match what $\mathsf{Sim}_{\mathsf{GLOS}}$ sends to them.

   If not, $\mathsf{Sim}_{\mathsf{GLOS}}$ sets $\mathsf{Comp}_j = 1$.
7. For each virtual server V_j, $\mathsf{Sim}_{\mathsf{GLOS}}$ faithfully emulates the random $\mathcal{O}$ to compute $\mathcal{O}(I_{j,\alpha} \| \sigma_{j,\alpha})$ for each corrupted server $S_{j,\alpha}$. For each honest server $S_{j,\beta}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random string (of the same length as an output from the random oracle) as $\mathcal{O}(I_{j,\beta} \| \sigma_{j,\beta})$.
8. $\mathsf{Sim}_{\mathsf{GLOS}}$ runs $\mathsf{Sim}_{\mathsf{GLOSGb}}$ (Figure 21).
9. $\mathsf{Sim}_{\mathsf{GLOS}}$ receives $\mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}$ from $\mathcal{F}_{\mathsf{GLOS}}$. $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive $(\mathsf{ver}, \mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}, \mathcal{T}_{\mathsf{ver}})$ from the corrupted parties. $\mathsf{Sim}_{\mathsf{GLOS}}$ checks whether the following hold:
   - The sets received from corrupted parties match the sets received from $\mathcal{F}_{\mathsf{GLOS}}$.
   - Let $\mathcal{T}_{\mathsf{cor}}$ be the set of virtual servers that are completely emulated by corrupted servers. It holds that $\mathcal{T}_{\mathsf{hon}} \cap (\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}}) = \emptyset$ and $|\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}| - |\mathcal{T}_{\mathsf{hon}}| \leq T$.
   - For all parties $V_j \in \mathcal{T}_{\mathsf{hon}}$, it holds that $\mathsf{Comp}_j = 0$.
   - For each Σ-sharing generated by a corrupted dealer in Π_{Share}, there exists a valid Σ-sharing $[\boldsymbol{v}]$ such that for each $V_j \in \mathcal{T}_{\mathsf{act}}$, the shares of $\langle([\boldsymbol{v}])^{V_j}\rangle$ (where $([\boldsymbol{v}])^{V_j}$ is V_j's share of $[\boldsymbol{v}]$) for the honest servers among $S_{j,1}, \ldots, S_{j,c}$ match the messages sent by the corrupted dealer in Π_{Share}, and the shares for corrupted servers match their committed inputs to $\Pi_{\mathsf{Gb}}(V_j)$. Similar for the $\Sigma^{(2)}, \Sigma^{(3)}$-sharings and the $\Sigma^{(2)}$-sharings with all-zero secrets.

   If true, $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to send $\mathsf{success}$ to all the corrupted parties.
10. If $\mathsf{success}$ has been sent, $\mathsf{Sim}_{\mathsf{GLOS}}$ runs $\mathsf{Sim}_{\mathsf{GLOSOutput}}$ (Figure 22). Otherwise, $\mathsf{Sim}_{\mathsf{GLOS}}$ sends $\perp$ to $\mathcal{F}_{\mathsf{GLOS}}$ as each honest server's output.
11. $\mathsf{Sim}_{\mathsf{GLOS}}$ outputs the adversary's view.

Figure 19: The simulator $\mathsf{Sim}_{\mathsf{GLOS}}$.

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

$\mathsf{Sim}_{\mathsf{GLOS}}$ follows the protocol to transform the circuit C to C', and then:

1. **Input Sharings.** For each batch of k input wires attached to an honest client $\mathtt{C}_i$ in circuit C', $\mathsf{Sim}_{\mathsf{GLOS}}$ randomly samples corrupted servers' shares of the additive sharing of each virtual server's shares of $[\boldsymbol{s}]$ and

emulates $\mathcal{F}_{\mathsf{emulate}}$ to send them to the corrupted servers with respect to sender $\mathtt{C}_i$. I.e., for each message m sent to a corrupted server, $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to send $(\mathtt{C}_i, m)$. For each batch of k input wires attached to a corrupted client $\mathtt{C}_i$ in circuit C', $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive each honest server $S_{j,\alpha}$'s share of the additive sharing of each honest virtual server V_j's shares of $[\boldsymbol{s}]$ from $\mathtt{C}_i$.

2. **Masks Sharings for Outputs.** For each batch of k output wires attached to an honest client $\mathtt{C}_i$ in circuit C', $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $[\boldsymbol{r}']^{(2)}$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send them to the corrupted servers with respect to sender $\mathtt{C}_i$. For each batch of k input wires attached to a corrupted client $\mathtt{C}_i$ in circuit C', $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive each honest server's share of the additive sharing of each honest virtual server V_j's shares of $[\boldsymbol{r}']^{(2)}$ from $\mathtt{C}_i$.
3. **Random $\Sigma^{(2)}$-Sharings.** For each $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{s}]^{(2)}_a$ shared by an honest server S, $\mathsf{Sim}_{\mathsf{GLOS}}$ randomly samples corrupted servers' shares of $\langle \boldsymbol{s}^{V_j} \rangle$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send them to the corrupted servers with respect to sender S. For each $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{s}]^{(2)}_a$ shared by a corrupted server S, $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive each honest server's share of the additive sharing of each honest virtual server V_j's share of $[\boldsymbol{s}]^{(2)}_a$ from S.
4. **Zero Sharings.** For each $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{s}]^{(2)}_a$ with an all-zero secret shared by an honest server S, $\mathsf{Sim}_{\mathsf{GLOS}}$ randomly samples corrupted servers' shares of $\langle \boldsymbol{s}^{V_j} \rangle$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send them to the corrupted servers with respect to sender S. For each $\Sigma^{(2)}_{\times a}$-sharing $[\boldsymbol{s}]^{(2)}_a$ shared by a corrupted server S, $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive each honest server's share of the additive sharing of each honest virtual server V_j's share of $[\boldsymbol{s}]^{(2)}_a$ from S.
5. **Sharings 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, $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive each honest server $S_{j,\alpha}$'s shares of the additive sharings of V_j's shares of $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ for each $\beta = 1, \ldots, c$.
   - If the receiver R_i of the i-th reconstruction of Π_0 is an honest client, $\mathsf{Sim}_{\mathsf{GLOS}}$ randomly samples corrupted servers' shares of the additive sharings for each virtual server's shares of $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ for $\beta = 1, \ldots, c$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send them to the corrupted servers with respect to sender R_i.
   - If the receiver R_i of the i-th reconstruction of Π_0 is an honest virtual server V_j, $\mathsf{Sim}_{\mathsf{GLOS}}$ randomly samples corrupted servers' shares of the additive sharings for each virtual server's shares of $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},1}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send them to the corrupted servers with respect to sender $S_{j,\beta}$. Then, for each honest virtual server V_η, $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive honest servers' shares of the additive sharings of V_η's shares of $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ for each corrupted server $S_{j,\beta}$.

Figure 20: The simulator for the sharing phase.

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

Garbling Phase

Let $\mathcal{T}_{\mathsf{cor}}$ be the set of corrupted virtual servers. If $|\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}| \leq T$, then for each sharing among the virtual servers generated by an honest party during Π_{Share}, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples the shares for virtual servers in $\mathcal{T}_{\mathsf{ver}}$ based on the shares for corrupted virtual servers. After that, for each virtual server $V_j \in \mathcal{T}_{\mathsf{ver}}$:

1. For each additive sharing among $S_{j,1}, \ldots, S_{j,c}$ generated by an honest party during Π_{Share}, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples the honest servers' shares based on the secret and corrupted servers' shares.
2. $\mathsf{Sim}_{\mathsf{GLOS}}$ randomly samples each honest server $S_{j,i}$'s seed for generating the randomness for the protocol execution of $\Pi_{\mathsf{Gb}}(V_j)$.
3. For each ROT instance between two servers emulating V_j prepared by $\mathcal{F}_{\mathsf{GLOSbaseROT}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ faithfully emulates $\mathcal{F}_{\mathsf{GLOSbaseROT}}$ to generate honest servers' outputs from it.
4. Now, $\mathsf{Sim}_{\mathsf{GLOS}}$ already obtains all the honest servers' inputs to $\Pi_{\mathsf{Gb}}(V_j)$. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ honestly emulates the honest servers to interact with corrupted servers during $\Pi_{\mathsf{Gb}}(V_j)$.

For each virtual server V_j under either of the following two conditions:

– $|\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}| > T$;

– $V_j \notin \mathcal{T}_{\mathsf{ver}}$ and $|\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}| \leq T$,

$\mathsf{Sim}_{\mathsf{GLOS}}$ does the following.

1. **Local Computation.** For each corrupted server $S_{j,i}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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. **OT Extension.** During each execution of Π_{OText} between two corrupted servers, $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ faithfully. During each execution of Π_{OText} between an honest server $S_{j,\alpha}$ and a corrupted server $S_{j,\beta}$, whenever $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive a message m from $S_{j,\beta}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ set the corresponding message sent by $\mathcal{A}_{\mathsf{OText}}$ to m while running $\mathsf{Sim}_{\mathsf{OText}}$. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ receives from $\mathsf{Sim}_{\mathsf{OText}}$ $S_{j,\alpha}$'s next message during Π_{OText} and emulates $\mathcal{F}_{\mathsf{emulate}}$ deliver it to $S_{j,\beta}$ with respect to sender $S_{j,\alpha}$.
3. **Garbling Local Circuits.**
   (a) For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wires a, b and output wire o, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 corrupted server $S_{j,i}$.
   (b) For each wire w in $\mathtt{Circ}^{V_j}$ that is not an output wire of an output gate. $\mathsf{Sim}_{\mathsf{GLOS}}$ follows the protocol to compute $k_{w,1}^{S_{j,i}} = k_{w,0}^{S_{j,i}} \oplus \Delta^{S_{j,i}}$ for each corrupted server $S_{j,i}$.
   (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 honest server $S_{j,\alpha}$ and corrupted server $S_{j,\beta}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random κ-bit string as $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$, samples a random bit as $y^{(\alpha)} \oplus b^{(\beta,\alpha)}$, and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send them to $S_{j,\beta}$ with respect to sender $S_{j,\alpha}$. Then $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive $r_1^{(\beta,\alpha)} \oplus r_0^{(\beta,\alpha)} \oplus x^{(\beta)}$ and $y^{(\beta)} \oplus b^{(\alpha,\beta)}$ from $S_{j,\beta}$ and checks whether they are correctly computed with their committed inputs. If not, $\mathsf{Sim}_{\mathsf{GLOS}}$ sets $\mathsf{Comp}_j = 1$.
      ii. $\mathsf{Sim}_{\mathsf{GLOS}}$ honestly simulates the random oracle $\mathcal{O}$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send the RO outputs. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes each corrupted server $S_{j,i}$'s shares of $\langle \boldsymbol{A}_{g,1}^{V_j} \rangle, \langle \boldsymbol{A}_{g,2}^{V_j} \rangle, \langle \boldsymbol{A}_{g,3}^{V_j} \rangle, \langle \boldsymbol{A}_{g,4}^{V_j} \rangle$ based on their committed inputs, the randomly sampled messages sent from honest servers, and the outputs of $\mathcal{O}$.
   (d) For each output gate (with index $\mathsf{k} = (i-1)\ell^2 + a$ 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 honest server $S_{j,\alpha}$ and each corrupted server $S_{j,\beta}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random κ-bit string as $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$, samples a random bit as $y^{(\alpha)} \oplus b^{(\beta,\alpha)}$, and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send them to $S_{j,\beta}$ with respect to sender $S_{j,\alpha}$. Then $\mathsf{Sim}_{\mathsf{GLOS}}$ emulates $\mathcal{F}_{\mathsf{emulate}}$ to receive $r_1^{(\beta,\alpha)} \oplus r_0^{(\beta,\alpha)} \oplus x^{(\beta)}$ and $y^{(\beta)} \oplus b^{(\alpha,\beta)}$ from $S_{j,\beta}$ and checks whether they are correctly computed with their committed inputs. If not, $\mathsf{Sim}_{\mathsf{GLOS}}$ sets $\mathsf{Comp}_j = 1$.
      ii. $\mathsf{Sim}_{\mathsf{GLOS}}$ honestly simulates the random oracle $\mathcal{O}$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send the RO output. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes each corrupted server $S_{j,i}$'s shares of $\langle \mathbf{ct}_{w,0}^{V_j} \rangle, \langle \mathbf{ct}_{w,1}^{V_j} \rangle$ based on their committed inputs, the randomly sampled messages sent from honest servers, and the outputs of $\mathcal{O}$.
4. **Masking Input Wire Values.** For each input wire w of $\mathtt{Circ}^{V_j}$:
   (a) $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random bit as each honest server $S_{j,i}$'s share of $\langle x_w \oplus \lambda_w \rangle$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send it to the corrupted servers emulating V_j with respect to sender $S_{j,i}$.
   (b) $\mathsf{Sim}_{\mathsf{GLOS}}$ receives corrupted servers' shares of $\langle x_w \oplus \lambda_w \rangle$ and checks whether they match the inputs they committed. If not, $\mathsf{Sim}_{\mathsf{GLOS}}$ sets $\mathsf{Comp}_j = 1$. Otherwise, $\mathsf{Sim}_{\mathsf{GLOS}}$ reconstructs $x_w \oplus \lambda_w$.

Figure 21: The simulator for the garbling phase.

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

For each $V_j \in \mathcal{T}_{\mathsf{ver}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ sends each honest server $S_{j,i}$'s view during $\Pi_{\mathsf{Gb}}(V_j)$ to $\mathcal{F}_{\mathsf{GLOS}}$.

For each virtual server $V_j \in \mathcal{T}_{\mathsf{act}} \backslash (\mathcal{T}_{\mathsf{hon}} \cup \mathcal{T}_{\mathsf{ver}})$:

1. $\mathsf{Sim}_{\mathsf{GLOS}}$ randomly samples each honest server $S_{j,i}$'s shares of all the sharings generated by an honest party in the sharing phase based on corrupted servers' shares.
2. For each honest server $S_{j,i}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ randomly samples a κ-bit string as $\Delta^{S_{j,i}}$.
3. 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 $\mathtt{Circ}^{V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ randomly samples $\lambda_w^{S_{j,i}}, k_{w,0}^{S_{j,i}}$ for each honest server $S_{j,i}$.
4. For each input wire w of $\mathtt{Circ}^{V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes each honest server $S_{j,i}$'s share $\lambda_w^{S_{j,i}}$ of $\langle \lambda_w \rangle$ based on his shares of $\langle x_w \oplus \lambda_w \rangle$ and $\langle x_w \rangle$.
5. For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wires a, b and output wire o, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes
$$k_{o,0}^{S_{j,i}} \| \lambda_o^{S_{j,i}} = (k_{a,0}^{S_{j,i}} \| \lambda_a^{S_{j,i}}) \oplus (k_{b,0}^{S_{j,i}} \| \lambda_b^{S_{j,i}})$$
for each honest server $S_{j,i}$. This computation is performed gate by gate.
6. For each wire w in $\mathtt{Circ}^{V_j}$ that is not an output wire of an output gate, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes $k_{w,1}^{S_{j,i}} = k_{w,0}^{S_{j,i}} \oplus \Delta^{S_{j,i}}$ for each honest server $S_{j,i}$
7. For each execution of Π_{Mult} in the garbling phase, for each honest server $S_{j,\alpha}$ and each corrupted server $S_{j,\beta}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ already has $r_1^{(\alpha,\beta)} \oplus r_0^{(\alpha,\beta)} \oplus x^{(\alpha)}$, $(b^{(\alpha,\beta)}, r_{b^{(\alpha,\beta)}}^{(\alpha,\beta)})$, $y^{(\alpha)} \oplus r_{b^{(\beta,\alpha)}}^{(\beta,\alpha)}$, and $x^{(\alpha)}, y^{(\alpha)}$. With these values, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes $S_{j,\alpha}$'s output from $\mathcal{F}_{\mathsf{ROT}}(L)$.
8. For each AND gate in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes each honest server $S_{j,i}$'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.
9. For each output gate in $\mathtt{Circ}^{V_j}$ with input wire w, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes each honest server $S_{j,i}$'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. $\mathsf{Sim}_{\mathsf{GLOS}}$ knows all the honest servers' inputs to $\Pi_{\mathsf{Gb}}(V_j)$ and computes all the honest servers' views during $\Pi_{\mathsf{Gb}}(V_j)$.
10. $\mathsf{Sim}_{\mathsf{GLOS}}$ sends each honest server $S_{j,i}$'s outputs of $\Pi_{\mathsf{Gb}}(V_j)$ to $\mathcal{F}_{\mathsf{GLOS}}$.

$\mathsf{Sim}_{\mathsf{GLOS}}$ extracts the secrets of corrupted clients' input Σ-sharings (obtained in Step 9 of $\mathsf{Sim}_{\mathsf{GLOS}}$) as their inputs to the circuit C. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ sends them to $\mathcal{F}_{\mathsf{GLOS}}$ on behalf of the corrupted clients. $\mathsf{Sim}_{\mathsf{GLOS}}$ then receives corrupted clients' outputs of C from $\mathcal{F}_{\mathsf{GLOS}}$ and runs the simulator for the underlying protocol with corrupted clients' inputs and outputs of C. In this way, $\mathsf{Sim}_{\mathsf{GLOS}}$ obtains corrupted parties' views during the underlying protocol execution together with the $\Sigma^{(2)}$-sharings reconstructed in the online phase of it (refer to [GLOS25] for more details). In particular, for each $i = 1, \ldots, \mathsf{rec}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ learns the shares for each $V_j \in \mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{s}_i]^{(2)}$ using the simulator for the underlying protocol. $\mathsf{Sim}_{\mathsf{GLOS}}$ then finds a $\Sigma^{(2)}$-sharing as $[\boldsymbol{s}_i]^{(2)}$ for each $i = 1, \ldots, \mathsf{rec}$ such that the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ matches the output from the simulator for the underlying protocol. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ does the following:

1. If the receiver R_i of the i-th reconstruction is an honest client, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\alpha = 1, \ldots, \kappa$, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}]^{(3)} = [\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$ and the secret $\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}$ for each $\beta = 1, \ldots, c$.
2. If the receiver R_i of the i-th reconstruction is a virtual server V_j in $\mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random κ-bit string as $\boldsymbol{r}_{s_{i,\eta},\eta,\beta} = (r_{s_{i,\eta},\eta,\beta}^{(1)}, \ldots, r_{s_{i,\eta},\eta,\beta}^{(\kappa)})$ for each $\eta = 1, \ldots, k$ and each honest server $S_{j,\beta}$, where $s_{i,\eta}$ is the η-th bit of $\boldsymbol{s}_i$. Then, for $\alpha = 1, \ldots, \kappa$, let $\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)} = (r_{s_{i,1},1,\beta}^{(\alpha)}, r_{s_{i,2},2,\beta}^{(\alpha)}, \ldots, r_{s_{i,k},k,\beta}^{(\alpha)})$, $\mathsf{Sim}_{\mathsf{GLOS}}$ randomly samples the whole sharings $[\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}]^{(3)}$ based on the shares of the virtual servers not in $\mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}]^{(3)} = [\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$ and the secret $\boldsymbol{r}_{\boldsymbol{s}_i,\beta}^{(\alpha)}$.
3. For each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ and each $a = 1, \ldots, \ell^2$, let $s_{i,a}^{V_j}$ be the a-th bit of V_j's share of $[\boldsymbol{s}_i]$. $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{r}_{\boldsymbol{s}_i,1}^{(\alpha)}]^{(3)}$ generated by the honest clients and the first servers emulating virtual servers in $\mathcal{T}_{\mathsf{hon}}$ and each pair of $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)}, [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$

generated by other parties, where the computational process of each $Y^{S_{j,\beta}}_{(i-1)\ell^2+a,s_{i,a}^{V_j}}$ is the same as in the protocol with each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ generated by an honest client or an honest server $S_{j,\beta}$ for $V_j \in \mathcal{T}_{\mathsf{hon}}$ being regarded as $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\boldsymbol{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]$.

For each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$:

1. 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, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random bit as $v_w \oplus \lambda_w$ and a random κ-bit string $k^{S_{j,i}}_{w,v_w\oplus\lambda_w}$ for each honest server $S_{j,i}$. For each input wire of $\mathtt{Circ}^{V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random κ-bit string $k^{S_{j,i}}_{w,v_w\oplus\lambda_w}$ for each honest server $S_{j,i}$.
2. For each XOR gate in $\mathtt{Circ}^{V_j}$ with input wires a, b and output wire o, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes $k^{S_{j,i}}_{o,v_o\oplus\lambda_o} = k^{S_{j,i}}_{a,v_a\oplus\lambda_a} \oplus k^{S_{j,i}}_{b,v_b\oplus\lambda_b}$ and $v_o \oplus \lambda_o = (v_a \oplus \lambda_a) \oplus (v_b \oplus \lambda_b)$ gate by gate from the first layer for each honest server $S_{j,i}$.
3. For each AND gate g in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the ciphertext encrypted by $\{k^{S_{j,i}}_{a,v_a\oplus\lambda_a}, k^{S_{j,i}}_{b,v_b\oplus\lambda_b}\}_{i=1}^{c}$ by:
$$\Bigg(\bigoplus_{i=1}^{c}\big(\mathcal{O}(k^{S_{j,i}}_{a,v_a\oplus\lambda_a}\|(v_a\oplus\lambda_a)\|k^{S_{j,i}}_{b,v_b\oplus\lambda_b}\|(v_b\oplus\lambda_b)\|i\|j\|\beta\|g)\big)\Bigg) \oplus k^{S_{j,\beta}}_{o,v_o\oplus\lambda_o}\|(v_o\oplus\lambda_o).$$
Then $\mathsf{Sim}_{\mathsf{GLOS}}$ samples 3 random κ-bit strings as the other 3 ciphertexts for this gate g and each $\beta = 1, \ldots, c$.
4. 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 the underlying protocol. $\mathsf{Sim}_{\mathsf{GLOS}}$ sets the output wire value v_w to be the corresponding bit from the output of the simulator for the underlying protocol. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$.
5. For each output gate in $\mathtt{Circ}^{V_j}$ indexed $\mathsf{k} = 1, \ldots, \ell^2\mathsf{rec}$ with input wire w, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes
$$\mathsf{ct}^{S_{j,\beta}}_{w,v_w\oplus\lambda_w} = \Bigg(\bigoplus_{i=1}^{c}\mathcal{O}(k^{S_{j,i}}_{w,v_w\oplus\lambda_w}\|(v_w\oplus\lambda_w)\|i\|j\|\beta\|w)\Bigg) \oplus Y^{S_{j,\beta}}_{\mathsf{k},\lambda_w}.$$
Then $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random $c\ell\kappa$-bit string as the other ciphertext for this wire w and each $\beta = 1, \ldots, c$.
6. For each input wire w of an output gate in $\mathtt{Circ}^{V_j}$ for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples each honest server $S_{j,i}$'s share of $\langle\lambda_w\rangle$ based on the secret and corrupted servers' shares.
7. For each $i = 1, \ldots, \mathsf{rec}$, if R_i is a virtual server V_j in $\mathcal{T}_{\mathsf{hon}}$ and the η-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_η in R_i's circuit $\mathtt{Circ}^{V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes
$$\mathsf{ct}^{(i,\alpha)}_{j_\eta,s_{i,\eta}} = \mathcal{O}(\boldsymbol{r}_{s_{i,\eta},\eta,\alpha}\|s_{i,\eta}\|i\|\alpha\|\eta\|j_\eta) \oplus \big(k^{S_{j,\alpha}}_{w_{j_\eta},v_{w_{j_\eta}}\oplus\lambda_{w_{j_\eta}}}\|(v_{w_{j_\eta}}\oplus\lambda_{w_{j_\eta}})\big).$$
Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random κ-bit string as $\mathsf{ct}^{(i,\alpha)}_{j_\eta,1\oplus s_{i,\eta}}$.
8. For each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes each honest server $S_{j,i}$'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.
9. $\mathsf{Sim}_{\mathsf{GLOS}}$ sends honest parties' outputs from $\Pi_{\mathsf{Gb}}(V_j)$ to $\mathcal{F}_{\mathsf{GLOS}}$.

Figure 22: The simulator for generating outputs.

G.4 Security Proof

We prove that Π computationally securely computes $\mathcal{F}_{\mathsf{GLOS}}$. We show that any PPT environment $\mathcal{Z}$ cannot distinguish the output in the ideal world from that in the real world with a non-negligible probability using

hybrid arguments.

We construct the following hybrids:

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

$\mathbf{Hyb}_1$: 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, $\mathsf{Sim}_{\mathsf{GLOS}}$ first randomly generates the corrupted virtual servers' shares (without the secret), and then randomly samples honest virtual servers' shares based on the 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 does not affect the output distribution. Thus, $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ have the same output distribution.

$\mathbf{Hyb}_2$: In this hybrid, whenever an honest party generates an additive sharing for the servers who act as a virtual server, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 does not affect the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same output distribution.

$\mathbf{Hyb}_3$: In this hybrid, for each honest virtual server V_j, $\mathsf{Sim}_{\mathsf{GLOS}}$ additionally sets $\mathsf{Comp}_j = 0$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same output distribution.

$\mathbf{Hyb}_4$: In this hybrid, after the parties commit their inputs to $\Pi_{\mathsf{Gb}}(V_j)$, $\mathsf{Sim}_{\mathsf{GLOS}}$ additionally checks whether the following holds:

- For each corrupted server $S_{j,\alpha}$'s share generated by honest parties during Π_{Share}, the committed share matches what $\mathsf{Sim}_{\mathsf{GLOS}}$ sends to $S_{j,\alpha}$.
- For each ROT instance, the committed outputs from $\mathcal{F}_{\mathsf{GLOSbaseROT}}$ of corrupted parties match what $\mathsf{Sim}_{\mathsf{GLOS}}$ sends to them.

If not, $\mathsf{Sim}_{\mathsf{GLOS}}$ additionally sets $\mathsf{Comp}_j = 1$. This does not 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 sharing generated by a corrupted party in the sharing phase, $\mathsf{Sim}_{\mathsf{GLOS}}$ regards that all the corrupted servers correctly commit the shares they get in the garbling phase, and then $\mathsf{Sim}_{\mathsf{GLOS}}$ can reconstruct the shares of honest virtual servers for each 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, $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same output distribution.

$\mathbf{Hyb}_6$: In this hybrid, during the garbling phase, during each execution of $\Pi_{\mathsf{Gb}}(V_j)$ under either of the two following conditions:

- $|\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}| > T$;
- $V_j \notin \mathcal{T}_{\mathsf{ver}}$ and $|\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}| \leq T$,

$\mathsf{Sim}_{\mathsf{GLOS}}$ does not follow the description of $\mathcal{F}_{\mathsf{GLOSbaseROT}}$ to generate the base ROT instances for each pair of parties $(S_{j,\alpha}, S_{j,\beta})$ when $S_{j,\alpha}$ is honest and $S_{j,\beta}$ is corrupted. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ initializes an adversary $\mathcal{A}_{\mathsf{OText}}$ for Π_{OText} (running between $S_{j,\alpha}, S_{j,\beta}$) that sends the same messages to $\mathcal{F}_{\mathsf{ROT}}(\kappa)$ and runs $\mathsf{Sim}_{\mathsf{OText}}$ with this adversary. $\mathsf{Sim}_{\mathsf{GLOS}}$ then receives the corresponding output for $S_{j,\beta}$ from $\mathcal{F}_{\mathsf{ROT}}(\kappa)$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send it to $S_{j,\beta}$. Whenever $\mathsf{Sim}_{\mathsf{OText}}$ invokes $\mathcal{F}_{\mathsf{ROT}}(L)$ (where $L = (4c+1) \cdot G_A^{V_j} + 2c\ell W_O^{V_j}$), $\mathsf{Sim}_{\mathsf{GLOS}}$ faithfully emulates $\mathcal{F}_{\mathsf{ROT}}(L)$ to receive $S_{j,\beta}$'s output of $\mathcal{F}_{\mathsf{ROT}}(L)$ and send it to $\mathsf{Sim}_{\mathsf{GLOS}}$. Similarly, $\mathsf{Sim}_{\mathsf{GLOS}}$ does not follow the description of $\mathcal{F}_{\mathsf{GLOSbaseROT}}$ to generate the base ROT instances for each pair of parties $(S_{j,\alpha}, S_{j,\beta})$ when $S_{j,\beta}$ is honest and $S_{j,\alpha}$ is corrupted. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ initializes an adversary $\mathcal{A}_{\mathsf{OText}}$ for Π_{OText} (running between $S_{j,\alpha}, S_{j,\beta}$) that sends the same messages to $\mathcal{F}_{\mathsf{ROT}}(\kappa)$ and runs $\mathsf{Sim}_{\mathsf{OText}}$ with this adversary. $\mathsf{Sim}_{\mathsf{GLOS}}$ then receives the corresponding output for $S_{j,\alpha}$ from $\mathcal{F}_{\mathsf{ROT}}(\kappa)$ and emulates $\mathcal{F}_{\mathsf{emulate}}$ to send it to $S_{j,\alpha}$. Whenever $\mathsf{Sim}_{\mathsf{OText}}$ invokes $\mathcal{F}_{\mathsf{ROT}}(L)$, $\mathsf{Sim}_{\mathsf{GLOS}}$ faithfully emulates $\mathcal{F}_{\mathsf{ROT}}(L)$ to receive $S_{j,\alpha}$'s output of $\mathcal{F}_{\mathsf{ROT}}(L)$ and send it to $\mathsf{Sim}_{\mathsf{GLOS}}$.

Moreover, $\mathsf{Sim}_{\mathsf{GLOS}}$ does not follow the protocol to run the OT extension protocol Π_{OText} for each pair of parties $(S_{j,\alpha}, S_{j,\beta})$ where one of them is corrupted, and the other is honest. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ runs the simulator $\mathsf{Sim}_{\mathsf{OText}}$ to generate the view of the corrupted server during Π_{OText}. The honest server's output from Π_{OText} is no longer computed from the protocol execution. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ directly emulates $\mathcal{F}_{\mathsf{ROT}}(L)$ faithfully to generate it. For the view of the honest party during Π_{OText} excluding his output, $\mathsf{Sim}_{\mathsf{GLOS}}$ does not use it during the simulation, so $\mathsf{Sim}_{\mathsf{GLOS}}$ no longer generates it.

Note that for these executions of Π_{OText}, we do not need honest servers' views during Π_{OText} (excluding the output) for further computation, so not generating them will not affect the output distribution. Therefore, we only change the way of generating honest servers' outputs of these executions of Π_{OText} together with the corrupted servers' views during Π_{OText}. From the security of Π_{OText}, the information generated in the two different ways is computationally indistinguishable. Now, we argue that $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ are computationally indistinguishable.

For the sake of contradiction, assume that $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ are computationally distinguishable. By the standard hybrid argument, there exists a PPT algorithm $\mathcal{D}$ that can distinguish the output distributions before and after we change honest servers' outputs and the corrupted servers' views of these executions of Π_{OText} to those simulated by running $\mathsf{Sim}_{\mathsf{OText}}$ and emulating $\mathcal{F}_{\mathsf{ROT}}(L)$ with a non-negligible probability. Now we construct a PPT algorithm $\mathcal{D}'$ that breaks the UC security of Π_{OText}. Concretely, $\mathcal{D}'$ receives honest servers' outputs and the corrupted servers' views of these executions of Π_{OText} from the challenger, which is either the real outputs and views from protocol execution or the simulated information. Then $\mathcal{D}'$ runs Sim with the change that the views and outputs are replaced by the information received from the challenger. Finally, $\mathcal{D}'$ uses $\mathcal{D}$ to distinguish whether the string received from the challenger is generated from a real execution or simulated. Note that the advantage of $\mathcal{D}'$ is the same as that of $\mathcal{D}$, which breaks UC security of Π_{OText}. This leads to a contradiction. Thus, the distributions of $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ are computationally indistinguishable.

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

$\mathbf{Hyb}_8$: In this hybrid, during the garbling phase, for each execution of Π_{Mult} during $\Pi_{\mathsf{Gb}}(V_j)$ under the same condition as in $\mathbf{Hyb}_6$, for each honest server $S_{j,\alpha}$ and each corrupted server $S_{j,\beta}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ additionally checks whether $r_1^{(\beta,\alpha)} \oplus r_0^{(\beta,\alpha)} \oplus x^{(\beta)}$ and $y^{(\beta)} \oplus b^{(\alpha,\beta)}$ from $S_{j,\beta}$ are correctly computed with their committed inputs. If not, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 under the same condition as in $\mathbf{Hyb}_6$ and each input wire w of $\mathtt{Circ}^{V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't follow the protocol to compute each honest server $S_{j,i}$'s share of $\langle x_w \oplus \lambda_w \rangle$. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random bit as $S_{j,i}$'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 corrupted servers' shares of $\langle x_w \oplus \lambda_w \rangle$, $\mathsf{Sim}_{\mathsf{GLOS}}$ additionally checks whether they match the inputs they committed. If not, $\mathsf{Sim}_{\mathsf{GLOS}}$ sets $\mathsf{Comp}_j = 1$. Since $S_{j,i}$'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, after emulating $\mathcal{F}_{\mathsf{emulate}}$ to receive $(\mathsf{ver}, \mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}, \mathcal{T}_{\mathsf{ver}})$ from the corrupted parties, $\mathsf{Sim}_{\mathsf{GLOS}}$ does not follow the description of $\mathcal{F}_{\mathsf{emulate}}$ to verify the execution of each Π_{Gb}. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$

verifies whether $\mathsf{Comp}_j = 0$. Note that Comp_j is set to 1 when $\Pi_{\mathsf{Gb}}(V_j)$ is not honestly executed by corrupted parties. The only difference is that $\mathsf{Sim}_{\mathsf{GLOS}}$ no longer checks the honest servers' execution. Since honest parties always follow the protocol, we only change the way of this verification without changing its result. Thus, $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ have the same output distribution.

$\mathbf{Hyb}_{11}$: In this hybrid, for each sharing generated by an honest party during Π_{Share}, $\mathsf{Sim}_{\mathsf{GLOS}}$ does not generate them during Π_{Share}. Instead:

- If $|\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}| \leq T$, for honest virtual servers in $\mathcal{T}_{\mathsf{ver}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ generates their shares of these sharings based on the corrupted virtual servers' shares at the beginning of the garbling phase. Then:
  - If it holds that $\mathcal{T}_{\mathsf{hon}} \cap (\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}}) = \emptyset$ and $|\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}| - |\mathcal{T}_{\mathsf{hon}}| \leq T$, for honest virtual servers of $\mathcal{T}_{\mathsf{act}} \backslash (\mathcal{T}_{\mathsf{hon}} \cup \mathcal{T}_{\mathsf{ver}})$, $\mathsf{Sim}_{\mathsf{GLOS}}$ generates their shares of these sharings based on the shares for virtual servers in $\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}$ while generating their outputs from $\Pi_{\mathsf{Gb}}(V_j)$. For other honest virtual servers, $\mathsf{Sim}_{\mathsf{GLOS}}$ generates their shares based on the previously sampled virtual servers' shares and the secret while generating their outputs from $\Pi_{\mathsf{Gb}}(V_j)$.
  - Otherwise, $\mathsf{Sim}_{\mathsf{GLOS}}$ generates all other honest virtual servers' shares based on corrupted virtual servers' shares and the secret at the beginning of the garbling phase.
- Otherwise, $\mathsf{Sim}_{\mathsf{GLOS}}$ generates all the honest virtual servers' shares based on corrupted virtual servers' shares and the secret at the beginning of the garbling phase.

Finally, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples the shares of each honest server $S_{j,i}$'s shares of the additive sharings of these shares based on corrupted servers' shares and the secrets. Under the condition $\mathcal{T}_{\mathsf{hon}} \cap (\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}}) = \emptyset$ and $|\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}| - |\mathcal{T}_{\mathsf{hon}}| \leq T$, for all these sharings, the set of corrupted servers' shares and the shares of honest virtual servers of $(\mathcal{T}_{\mathsf{act}} \cup \mathcal{T}_{\mathsf{ver}}) \backslash \mathcal{T}_{\mathsf{hon}}$ are independent of the secret because of the restriction $|\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}| - |\mathcal{T}_{\mathsf{hon}}| \leq T$, first sampling the shares of them and then sampling the shares for other honest virtual servers won't affect the output distribution. In other cases, we simply delay the generation of honest servers' shares without changing their distributions. Besides, the shares for each honest server $S_{j,i}$ emulating each virtual server V_j in $\mathcal{T}_{\mathsf{hon}}$ are not used in the simulation before the evaluation phase. Therefore, delaying the generation won't affect the output distribution. Thus, $\mathbf{Hyb}_{11}$ and $\mathbf{Hyb}_{10}$ have the same output distribution.

$\mathbf{Hyb}_{12}$: In this hybrid, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples the local randomness inputs to $\Pi_{\mathsf{Gb}}(V_j)$ of each honest server $S_{j,i}$ in emulating each $V_j \in \mathcal{T}_{\mathsf{hon}}$ while generating his outputs (rather than generating it while committing $I_{j,i}$) and the computation of $S_{j,i}$ is delayed to be performed then. For the commitment $\mathcal{O}(I_{j,i} \| \sigma_{j,i})$, $\mathsf{Sim}_{\mathsf{GLOS}}$ does not faithfully emulate $\mathcal{F}_{\mathsf{emulate}}$ to generate it. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ first generates a random string as the commitment. Moreover, the random κ-bit string $\sigma_{j,i}$ is no longer used in the simulation, so $\mathsf{Sim}_{\mathsf{GLOS}}$ does not sample it in future hybrids.

Since the randomness (including $\sigma_{j,i}$) for each honest server $S_{j,i}$ emulating each virtual server V_j in $\mathcal{T}_{\mathsf{hon}}$ are not used in the simulation before the evaluation phase (excluding the commitments), delaying the generation of them won't affect the output distribution if the distribution of the commitments does not change. Note that $\mathbf{O}(I_{j,i} \| \sigma_{j,i})$ is uniformly random if $\mathcal{A}$ and honest servers have not queried $I_{j,i} \| \sigma_{j,i}$ before. Therefore, the distribution only happens when $I_{j,i} \| \sigma_{j,i}$ has been queried. Since $\sigma_{j,i}$ is not used anywhere else, we can sample it after the commitment is sampled and then check whether $I_{j,i} \| \sigma_{j,i}$ is queried before. Note that is a κ-bit random string, the probability that the query $I_{j,i} \| \sigma_{j,i}$ after sampling $\sigma_{j,i}$ matches the $\mathsf{poly}(\kappa)$ previous RO queries is no more than $\mathsf{poly}(\kappa)/2^{\kappa}$, which is negligible, so the distribution only changes with a negligible probability. Then, $\sigma_{j,i}$ is not used in the simulation anymore, so not generating it does not change the output distribution. Thus, the distributions of $\mathbf{Hyb}_{12}$ and $\mathbf{Hyb}_{11}$ are computationally indistinguishable.

$\mathbf{Hyb}_{13}$: In this hybrid, if $\mathsf{Sim}_{\mathsf{GLOS}}$ has emulated $\mathcal{F}_{\mathsf{emulate}}$ to output $\mathsf{success}$, for each wire w in each honest virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$'s local circuit $\mathtt{Circ}^{V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ additionally follows the execution of the underlying protocol to compute the value v_w of w. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples the whole sharing $[\boldsymbol{s}_i]^{(2)}$ from the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{s}_i]^{(2)}$. If the receiver is an honest client, $\mathsf{Sim}_{\mathsf{GLOS}}$ uses them to compute shares for virtual servers out of $\mathcal{T}_{\mathsf{hon}}$ 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 $\beta = 1, \ldots, c$. If the receiver is a virtual server in $\mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ uses them to compute shares for virtual servers out of $\mathcal{T}_{\mathsf{hon}}$

of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)} + [\boldsymbol{s}_i]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ for each $\alpha = 1, \ldots, \kappa$ and each honest server $S_{j,\beta}$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{13}$ and $\mathbf{Hyb}_{12}$ have the same output distribution.

$\mathbf{Hyb}_{14}$: In this hybrid, if $\mathsf{Sim}_{\mathsf{GLOS}}$ has emulated $\mathcal{F}_{\mathsf{emulate}}$ to output $\mathsf{success}$, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$:

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

   - If R_i is an honest client, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't generate the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}$ first and then computes each share for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ by himself. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ 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 $\mathcal{T}_{\mathsf{hon}}$, for each honest server $S_{j,\beta}$ $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't generate the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}$ first and then computes their shares of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ by himself. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}$. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the shares for virtual servers out of $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}$ based on their shares of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_i,\beta}]^{(3)}, [\boldsymbol{s}_i]^{(2)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$.
   - If R_i is a virtual server out of $\mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ just follows the protocol to receive honest servers' shares from corrupted parties.

   Then $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\mathcal{T}_{\mathsf{hon}}$ 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 $Y^{S_{j,\beta}}_{(i-1)\ell^2+a,1\oplus s^{V_j}_{i,a}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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, for each honest server $S_{j,i}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random bit as $v_w \oplus \lambda_w$ and a random κ-bit string as $k^{S_{j,i}}_{w,v_w\oplus\lambda_w}$. For each input wire of $\mathtt{Circ}^{V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random κ-bit string as $k^{S_{j,i}}_{w,v_w\oplus\lambda_w}$. Then $\mathsf{Sim}_{\mathsf{GLOS}}$ computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$ for all these wires. After all the ciphertexts are generated, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random κ-bit string as $\Delta^{S_{j,i}}$. Then, for each wire w in $\mathtt{Circ}^{V_j}$ that is not an output wire of an output gate, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes $k^{S_{j,i}}_{w,1\oplus v_w\oplus\lambda_w} = k^{S_{j,i}}_{w,v_w\oplus\lambda_w} \oplus \Delta^{S_{j,i}}$.

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

4. 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 each honest server $S_{j,\alpha}$ computes his shares of the additive sharings of $\mathsf{ct}^{S_{j,\beta}}_{w,i_2}$ for each $\beta = 1, \ldots, c$, $\mathsf{Sim}_{\mathsf{GLOS}}$ checks whether the query to the random oracle $\mathcal{O}$ has been queried before. If true, $\mathsf{Sim}_{\mathsf{GLOS}}$ aborts the simulation. Otherwise, $\mathsf{Sim}_{\mathsf{GLOS}}$ adds the query to Q.

5. For each AND gate g in $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't follow the protocol to compute each honest server $S_{j,i}$'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, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples 3 random κ-bit strings as the ciphertexts

and computes $S_{j,i}$'s shares of the additive sharings of all the ciphertexts based on the secrets and corrupted servers' shares. $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the output of $\mathcal{O}$ to the queries that are used to generate these ciphertexts based on the random strings and the wire labels of wire o.

6. For each output gate in $\mathtt{Circ}^{V_j}$ indexed $1, \ldots, \ell^2\mathsf{rec}$ with input wire w, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't follow the protocol to compute each honest server $S_{j,i}$'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, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random $c\ell\kappa$-bit string as the ciphertexts and computes $S_{j,i}$'s shares of the additive sharings of all the ciphertexts based on the secrets and corrupted servers' shares. $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the output of $\mathcal{O}$ to the queries that are used to generate these ciphertexts based on the random strings and the output labels.

7. For each input wire w of an output gate in $\mathtt{Circ}^{V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't follow the protocol to compute each honest server $S_{j,i}$'s share of $\langle\lambda_w\rangle$. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 the underlying protocol and the β-th bit of $\boldsymbol{s}_i$ is used as an input wire with index j_β in $\mathtt{Circ}^{V_j}$, for each honest server $S_{j,\alpha}$ $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't follow the protocol to compute $\mathsf{ct}^{(i,\alpha)}_{j_\beta,1\oplus s_{i,\alpha}}$. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random κ-bit string as $\mathsf{ct}^{(i,\alpha)}_{j_\beta,1\oplus s_{i,\alpha}}$.

To prove that the distributions of $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{13}$ are computationally indistinguishable, we additionally construct the following hybrids between $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{13}$.

$\mathbf{Hyb}_{14.0}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes each honest server $S_{j,i}$'s shares of the garbled circuit of $\mathtt{Circ}^{V_j}$ by computing the garbled sub-circuits $\mathtt{Circ}^{V_j}_1, \ldots, \mathtt{Circ}^{V_j}_{\mathsf{rec}}$ in order. This doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{14.0}$ and $\mathbf{Hyb}_{13}$ have the same distribution.

$\mathbf{Hyb}_{14.1.1}$: In this hybrid, $\mathsf{Sim}_{\mathsf{GLOS}}$ additionally computes the share for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{s}_1]^{(2)}$ by using the input labels associated with the input of V_j to evaluate the garbled gates of $\mathtt{Circ}^{V_j}_1$. Here, the inputs of V_j (excluding the randomness) are shared by additive sharings among $S_{j,1}, \ldots, S_{j,c}$, and all the secrets of these sharings have been computed by $\mathsf{Sim}_{\mathsf{GLOS}}$. 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 computation process of S_j's share of $[\boldsymbol{s}_1]^{(2)}$ in the underlying protocol is identical to $\mathtt{Circ}^{V_j}_1$. Since $V_j \in \mathcal{T}_{\mathsf{hon}}$, 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 the underlying protocol.

In addition, if R_1 is an honest client, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't generate the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},\beta}]^{(3)}$ first and then computes the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}]^{(3)}$ by himself. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\mathcal{T}_{\mathsf{hon}}$ 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, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ 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 $\mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't generate the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{0},1}]^{(3)}$ first and then computes the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,1}]^{(3)}$ by himself. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,1}]^{(3)}$ and the secret $\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,1}$. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ 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 $V_j \in \mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\mathcal{T}_{\mathsf{hon}}$ 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 $Y^{S_{j,\beta}}_{a,1\oplus s_{1,a}^{V_j}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ still follows the protocol to compute it.

Since the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}, [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ generated by the honest clients and the first servers emulating virtual servers in $\mathcal{T}_{\mathsf{hon}}$ are sampled randomly based on the shares for virtual servers out of $\mathcal{T}_{\mathsf{hon}}$ and the secret in the last hybrid, their shares of $[\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta}]^{(3)} = [\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)} + [\boldsymbol{s}_1]^{(2)} \otimes [\boldsymbol{r}^{(\alpha)}_{\mathbf{1},\beta} - \boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]$ are also random based on the shares for virtual servers out of $\mathcal{T}_{\mathsf{hon}}$ and the secret. Therefore, we only change the order of generating the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}^{(\alpha)}_{\boldsymbol{s}_1,\beta}]^{(3)}$ and $[\boldsymbol{r}^{(\alpha)}_{\mathbf{0},\beta}]^{(3)}$ generated by an honest party without changing their distributions. Thus, $\mathbf{Hyb}_{14.1.1}$ and $\mathbf{Hyb}_{14.0}$ have the same distribution.

$\mathbf{Hyb}_{14.1.2}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't follow the protocol to compute each honest server $S_{j,i}$'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, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random bit as $v_w \oplus \lambda_w$ and a random κ-bit string as $k^{S_{j,i}}_{w,v_w\oplus\lambda_w}$. For each input wire of $\mathtt{Circ}^{V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random κ-bit string as $k^{S_{j,i}}_{w,v_w\oplus\lambda_w}$. Then $\mathsf{Sim}_{\mathsf{GLOS}}$ 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, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes $k^{S_{j,i}}_{w,1\oplus v_w\oplus\lambda_w} = k^{S_{j,i}}_{w,v_w\oplus\lambda_w} \oplus \Delta^{S_{j,i}}$. 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, $\mathbf{Hyb}_{14.1.2}$ and $\mathbf{Hyb}_{14.1.1}$ have the same output distribution.

$\mathbf{Hyb}_{14.1.3}$: In this hybrid, $\mathsf{Sim}_{\mathsf{GLOS}}$ maintains a set Q. For each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, 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 $\{k^{S_{j,i}}_{a,v_a\oplus\lambda_a}, k^{S_{j,i}}_{b,v_b\oplus\lambda_b}\}_{i=1}^{c}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ checks whether each query of each honest server $S_{j,\alpha}$ to the random oracle $\mathcal{O}$ has been queried before. If true, $\mathsf{Sim}_{\mathsf{GLOS}}$ aborts the simulation. Otherwise, $\mathsf{Sim}_{\mathsf{GLOS}}$ adds the query to Q. 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 $\Delta^{S_{j,\alpha}}$ are generated. Since each query made to the random oracle by $S_{j,\alpha}$ contains either of the κ-bit strings $k^{S_{j,\alpha}}_{a,1\oplus v_a\oplus\lambda_a}$ or $k^{S_{j,\alpha}}_{b,1\oplus v_b\oplus\lambda_b}$ with $k^{S_{j,\alpha}}_{a,1\oplus v_a\oplus\lambda_a} - k^{S_{j,\alpha}}_{a,v_a\oplus\lambda_a} = k^{S_{j,\alpha}}_{b,1\oplus v_b\oplus\lambda_b} - k^{S_{j,\alpha}}_{b,v_b\oplus\lambda_b} = \Delta^{S_{j,\alpha}}$ which is uniformly random, the probability that each query made by the adversary is one of the queries made by the honest server is negligible. Taking the union bound of all the $\mathsf{poly}(\kappa)$ queries made by the adversary, the probability that some query has been queried (either by the honest server or by the adversary) is negligible. Thus, the distributions of $\mathbf{Hyb}_{14.1.3}$ and $\mathbf{Hyb}_{14.1.2}$ are computationally indistinguishable.

$\mathbf{Hyb}_{14.1.4}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, 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 $\{k^{S_{j,i}}_{a,v_a\oplus\lambda_a}, k^{S_{j,i}}_{b,v_b\oplus\lambda_b}\}_{i=1}^{c}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples random κ-bit strings as the ciphertexts and computes each honest server $S_{j,\alpha}$'s shares of the additive sharings of the ciphertexts based on the secrets and corrupted servers' shares. While emulating $\mathcal{O}$, for each $i_0, i_1 \in \{0,1\}$ such that $(i_0, i_1) \neq (v_a \oplus \lambda_a, v_b \oplus \lambda_b)$, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the output of $\mathcal{O}(k^{S_{j,\alpha}}_{a,i_0} \| i_0 \| k^{S_{j,\alpha}}_{b,i_1} \| i_1 \| \alpha \| j \| i \| g)$ for each $i = 1, \ldots, c$ based on the random strings and the wire labels of wire o. Note that the only difference between $\mathbf{Hyb}_{14.1.4}$ and $\mathbf{Hyb}_{14.1.3}$ is the way we decide the output for queries in Q. 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 $\mathsf{Sim}_{\mathsf{GLOS}}$ does not abort the simulation, queries in Q have not been queried before. Thus, $\mathbf{Hyb}_{14.1.4}$ and $\mathbf{Hyb}_{14.1.3}$ have the same output distribution.

$\mathbf{Hyb}_{14.1.5}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, for each honest server $S_{j,\alpha}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ changes the order of sampling random κ-bit strings as the ciphertexts of this gate except those computed with $\{k^{S_{j,i}}_{a,v_a\oplus\lambda_a}, k^{S_{j,i}}_{b,v_b\oplus\lambda_b}\}_{i=1}^{c}$ and sampling $\Delta^{S_{j,\alpha}}$ to decide the queries to $\mathcal{O}$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{14.1.5}$ and $\mathbf{Hyb}_{14.1.4}$ have the same output distribution.

$\mathbf{Hyb}_{14.1.6}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, each input wire w of an output gate in $\mathtt{Circ}_1^{V_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, while computing each honest server $S_{j,\alpha}$'s share of $\langle \mathsf{ct}^{S_{j,\beta}}_{w,i_2} \rangle$ for each $\beta = 1, \ldots, c$, $\mathsf{Sim}_{\mathsf{GLOS}}$ checks whether the query made by $S_{j,\alpha}$ to the random oracle $\mathcal{O}$ has been queried before. If true, $\mathsf{Sim}_{\mathsf{GLOS}}$ aborts the simulation. Otherwise, $\mathsf{Sim}_{\mathsf{GLOS}}$ adds the query to Q. Note that each query made to the random oracle contains a κ-bit string $k^{S_{j,\alpha}}_{w,1\oplus v_w\oplus\lambda_w}$ with $k^{S_{j,\alpha}}_{w,1\oplus v_w\oplus\lambda_w} - k^{S_{j,\alpha}}_{w,v_w\oplus\lambda_w} = \Delta^{S_{j,\alpha}}$,

for the same reason in $\mathbf{Hyb}_{14.1.3}$, the probability that some query has been queried is negligible. Thus, the distributions of $\mathbf{Hyb}_{14.1.6}$ and $\mathbf{Hyb}_{14.1.5}$ are computationally indistinguishable.

$\mathbf{Hyb}_{14.1.7}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, 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 $i_2 \neq v_w \oplus \lambda_w$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random $c\ell\kappa$-bit string as the ciphertext $\mathsf{ct}_{w,i_2}^{S_{j,\beta}}$ and samples each honest server's shares of the additive sharing of the ciphertext based on the secret and corrupted servers' shares. While emulating $\mathcal{O}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the output based on the random strings and the output labels. Note that the only difference between $\mathbf{Hyb}_{14.1.7}$ and $\mathbf{Hyb}_{14.1.6}$ is the way we decide the output for queries in Q. 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 $\mathsf{Sim}_{\mathsf{GLOS}}$ does not abort the simulation, queries in Q have not been queried before. Thus, $\mathbf{Hyb}_{14.1.7}$ and $\mathbf{Hyb}_{14.1.6}$ have the same output distribution.

$\mathbf{Hyb}_{14.1.8}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, for each input wire w of an output gate in $\mathtt{Circ}_1^{V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't follow the protocol to compute each honest server $S_{j,i}$'s share of $\langle\lambda_w\rangle$. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes it based on λ_w and the corrupted servers' shares of $\langle\lambda_w\rangle$. Since $S_{j,i}$'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 $S_{j,i}$'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 $S_{j,i}$'s share of $\langle\lambda_w\rangle$. Thus, $\mathbf{Hyb}_{14.1.8}$ and $\mathbf{Hyb}_{14.1.7}$ have the same output distribution.

$\mathbf{Hyb}_{14.1.9}$: In this hybrid, if the receiver R_1 of $[\boldsymbol{s}_1]^{(2)}$ is a virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ and the β-th bit of $\boldsymbol{s}_1$ is used as an input wire with index j_β in $\mathtt{Circ}^{V_j}$, when each honest server $S_{j,i}$ computes $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,i)}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ checks whether the query to the random oracle $\mathcal{O}$ has been queried before. If true, $\mathsf{Sim}_{\mathsf{GLOS}}$ aborts the simulation. Otherwise, $\mathsf{Sim}_{\mathsf{GLOS}}$ adds the query to Q. Since $\boldsymbol{r}_{1\oplus s_{1,\beta},\beta,i}$ 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 $\mathbf{Hyb}_{14.1.9}$ and $\mathbf{Hyb}_{14.1.8}$ are computationally indistinguishable.

$\mathbf{Hyb}_{14.1.10}$: In this hybrid, if the receiver R_1 of $[\boldsymbol{s}_1]^{(2)}$ is a virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ and the β-th bit of $\boldsymbol{s}_1$ is used as an input wire with index j_β in $\mathtt{Circ}^{V_j}$, for each honest server $S_{j,i}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't follow the protocol to compute $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,i)}$. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random κ-bit string as $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,i)}$. While emulating $\mathcal{O}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the output of $\mathcal{O}(\boldsymbol{r}_{1\oplus s_{1,\beta},\beta}\|s_{1,\beta}\|1\|i\|\beta\|j_\beta)$ based on $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,i)}$ and $k_{w_{j_\beta},1\oplus v_{w_{j_\beta}}^{S_{j,i}}\oplus\lambda_{w_{j_\beta}}}\|(1\oplus v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}})$. Note that the only difference between $\mathbf{Hyb}_{14.1.10}$ and $\mathbf{Hyb}_{14.1.9}$ is the way we decide the output for queries in Q. Since $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,i)}$ is randomly sampled, $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,i)} \oplus m$ is also uniformly random for any κ-bit string m. In particular, when $\mathsf{Sim}_{\mathsf{GLOS}}$ does not abort the simulation, queries in Q have not been queried before. Thus, $\mathbf{Hyb}_{14.1.10}$ and $\mathbf{Hyb}_{14.1.9}$ have the same output distribution.

$\mathbf{Hyb}_{14.1.11}$: In this hybrid, for virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, for each honest server $S_{j,i}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ changes the order of sampling a random κ-bit string as the ciphertext $\mathsf{ct}_{j_\beta,1\oplus s_{1,\beta}}^{(1,i)}$ and sampling $\boldsymbol{r}_{1\oplus s_{1,\beta},\beta,i}$ to decide the queries to $\mathcal{O}$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{14.1.11}$ and $\mathbf{Hyb}_{212.1.10}$ have the same output distribution.

$\mathbf{Hyb}_{14.1.12}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ and each honest server $S_{j,\alpha}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ changes the order of sampling random $c\ell\kappa$-bit strings as the ciphertexts $\mathsf{ct}_{w,i_2}^{S_{j,\beta}}$ 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 $\Delta^{S_{j,\alpha}}$ to decide the queries to $\mathcal{O}$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{14.1.12}$ and $\mathbf{Hyb}_{14.1.11}$ have the same output distribution.

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

Similarly, for each $\gamma = 2,\ldots,\mathsf{rec}$ we can define $\mathbf{Hyb}_{14.\gamma.1},\ldots,\mathbf{Hyb}_{14.\gamma.12}$.

$\mathbf{Hyb}_{14.\gamma.1}$: In this hybrid, $\mathsf{Sim}_{\mathsf{GLOS}}$ additionally computes the share for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{s}_\gamma]^{(2)}$ by using the input labels associated with the input of V_j to evaluate the garbled gates of $\mathtt{Circ}_\gamma^{V_j}$. Here, the inputs of V_j (excluding the randomness) are shared by additive sharings among $S_{j,1},\ldots,S_{j,c}$, and all the secrets of these sharings have been computed by $\mathsf{Sim}_{\mathsf{GLOS}}$. We use these secrets as the input and preprocessing data to compute the input labels. We denote the result of the computation be $\overline{[\boldsymbol{s}_\gamma]^{(2)}}$. 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 the underlying protocol with $\mathtt{Circ}_\gamma^{V_j}$, for the same reason in $\mathbf{Hyb}_{14.1.1}$, 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, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't generate the honest virtual servers' shares of each $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)}$ first and then computes the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}_{\boldsymbol{s}_\gamma,\beta}^{(\alpha)}]^{(3)}$ by himself. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ generates $[\boldsymbol{s}_\gamma]^{(2)}$ first and then samples the whole sharing $[\boldsymbol{r}_{\boldsymbol{s}_\gamma,\beta}^{(\alpha)}]^{(3)}$ based on the shares for virtual servers out of $\mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{r}_{\boldsymbol{s}_\gamma,\beta}^{(\alpha)}]^{(3)}$ and the secret $\boldsymbol{r}_{\boldsymbol{s}_\gamma,\beta}^{(\alpha)}$ for each $\beta = 1, \ldots, c$. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)}$ based on their shares of $[\boldsymbol{r}_{\boldsymbol{s}_\gamma,\beta}^{(\alpha)}]^{(3)}, [\boldsymbol{s}_\gamma]^{(2)}, [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$. Similarly, if R_γ is a virtual server in $\mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't generate the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}_{\boldsymbol{0},1}^{(\alpha)}]^{(3)}$ first and then computes the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of each $[\boldsymbol{r}_{\boldsymbol{s}_\gamma,1}^{(\alpha)}]^{(3)}$ by himself. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ generates $[\boldsymbol{s}_\gamma]^{(2)}$ first and then samples the whole sharing $[\boldsymbol{r}_{\boldsymbol{s}_\gamma,1}^{(\alpha)}]^{(3)}$ based on the shares for virtual servers out of $\mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{r}_{\boldsymbol{s}_\gamma,1}^{(\alpha)}]^{(3)}$ and the secret $\boldsymbol{r}_{\boldsymbol{s}_\gamma,1}^{(\alpha)}$. Then, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the shares for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ of $[\boldsymbol{r}_{\boldsymbol{0},1}^{(\alpha)}]^{(3)}$ based on their shares of $[\boldsymbol{r}_{\boldsymbol{s}_\gamma,1}^{(\alpha)}]^{(3)}, [\boldsymbol{s}_\gamma]^{(2)}, [\boldsymbol{r}_{\boldsymbol{1},1}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},1}^{(\alpha)}]$. Then for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ follows the computational process to compute each $Y_{a,s_{\gamma,a}}^{S_{j,\beta}\ V_j}$ in the protocol to compute each $Y_{a,s_{\gamma,a}}^{S_{j,\beta}\ V_j}$ with each $[\boldsymbol{r}_{\boldsymbol{s}_\gamma,\beta}^{(\alpha)}]^{(3)}$ generated by the honest clients and the first servers emulating virtual servers in $\mathcal{T}_{\mathsf{hon}}$ regarded as $[\boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]^{(3)} + [\boldsymbol{s}_\gamma]^{(2)} \otimes [\boldsymbol{r}_{\boldsymbol{1},\beta}^{(\alpha)} - \boldsymbol{r}_{\boldsymbol{0},\beta}^{(\alpha)}]$. For the other label $Y_{a,1\oplus s_{\gamma,a}}^{S_{j,\beta}\ V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ still follows the protocol to compute it. For the same reason in $\mathbf{Hyb}_{14.1.1}$, $\mathbf{Hyb}_{14.\gamma.1}$ and $\mathbf{Hyb}_{14.(\gamma-1).12}$ have the same distribution.

$\mathbf{Hyb}_{14.\gamma.2}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't follow the protocol to compute each honest server $S_{j,i}$'s shares of the garbled circuit for $\mathtt{Circ}_\gamma^{V_j}$. Instead, for each wire w in $\mathtt{Circ}_\gamma^{V_j}$ that is not an input wire of the circuit and is not an output wire of an XOR gate or an output gate, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random bit as $v_w \oplus \lambda_w$ and a random κ-bit string as $k_{w,v_w\oplus\lambda_w}^{S_{j,i}}$. Then $\mathsf{Sim}_{\mathsf{GLOS}}$ computes $\lambda_w = (v_w \oplus \lambda_w) \oplus v_w$ for these wires. For each wire w in $\mathtt{Circ}_\gamma^{V_j}$ that is not an output wire of an output gate, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes $k_{w,1\oplus v_w\oplus\lambda_w}^{S_{j,i}} = k_{w,v_w\oplus\lambda_w}^{S_{j,i}} \oplus \Delta^{S_{j,i}}$. For the same reason in $\mathbf{Hyb}_{14.1.2}$, $\mathbf{Hyb}_{14.\gamma.2}$ and $\mathbf{Hyb}_{14.\gamma.1}$ have the same output distribution.

$\mathbf{Hyb}_{14.\gamma.3}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, for each AND gate g in $\mathtt{Circ}_\gamma^{V_j}$ with input wire a, b, and for the ciphertexts of this gate except those computed with $\{k_{a,v_a\oplus\lambda_a}^{S_{j,i}}, k_{b,v_b\oplus\lambda_b}^{S_{j,i}}\}_{i=1}^c$, $\mathsf{Sim}_{\mathsf{GLOS}}$ checks whether each query of each honest server to the random oracle $\mathcal{O}$ has been queried before. If true, $\mathsf{Sim}_{\mathsf{GLOS}}$ aborts the simulation. Otherwise, $\mathsf{Sim}_{\mathsf{GLOS}}$ adds the query to Q. For the same reason in $\mathbf{Hyb}_{14.1.3}$, the distributions of $\mathbf{Hyb}_{14.\gamma.3}$ and $\mathbf{Hyb}_{14.\gamma.2}$ are computationally indistinguishable.

$\mathbf{Hyb}_{14.\gamma.4}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, for each AND gate g in $\mathtt{Circ}_\gamma^{V_j}$ with input wire a, b and output wire o, and for the ciphertexts of this gate except those computed with $\{k_{a,v_a\oplus\lambda_a}^{S_{j,i}}, k_{b,v_b\oplus\lambda_b}^{S_{j,i}}\}_{i=1}^c$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples random κ-bit strings as the ciphertexts and samples each honest server $S_{j,\alpha}$'s shares of the additive sharings of the ciphertexts based on the secrets and corrupted servers shares. While emulating $\mathcal{O}$, for each $i_0, i_1 \in \{0,1\}$ such that $(i_0, i_1) \neq (v_a \oplus \lambda_a, v_b \oplus \lambda_b)$, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the output of $\mathcal{O}(k_{a,i_0}^{S_{j,\alpha}} \| i_0 \| k_{b,i_1}^{S_{j,\alpha}} \| i_1 \| \alpha \| j \| i \| g)$ for each $i = 1, \ldots, c$ based on the random strings and the wire labels of wire o. For the same reason in $\mathbf{Hyb}_{14.1.4}$, $\mathbf{Hyb}_{14.\gamma.4}$ and $\mathbf{Hyb}_{14.\gamma.3}$ have the same output distribution.

$\mathbf{Hyb}_{14.\gamma.5}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ and each honest server $S_{j,\alpha}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\Delta^{S_{j,\alpha}}$ to decide the queries to $\mathcal{O}$. For the same reason in $\mathbf{Hyb}_{14.1.5}$, $\mathbf{Hyb}_{14.\gamma.5}$ and $\mathbf{Hyb}_{14.\gamma.4}$ have the same output distribution.

$\mathbf{Hyb}_{14.\gamma.6}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, for each input wire w of an output gate in $\mathtt{Circ}_\gamma^{V_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, while computing each honest server $S_{j,i}$'s share of $\langle \mathsf{ct}_{w,i_2}^{S_{j,\beta}} \rangle$ for each $\beta = 1, \ldots, c$, $\mathsf{Sim}_{\mathsf{GLOS}}$ checks whether the query made by $S_{j,i}$ to the random oracle $\mathcal{O}$ has been queried before. If true, $\mathsf{Sim}_{\mathsf{GLOS}}$ aborts the simulation. Otherwise, $\mathsf{Sim}_{\mathsf{GLOS}}$ adds the query to

Q. For the same reason in $\mathbf{Hyb}_{14.1.6}$, the distributions of $\mathbf{Hyb}_{14.\gamma.6}$ and $\mathbf{Hyb}_{14.\gamma.5}$ are computationally indistinguishable.

$\mathbf{Hyb}_{14.\gamma.7}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, for each input wire w of an output gate in $\mathtt{Circ}_\gamma^{V_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random $c\ell\kappa$-bit string as the ciphertext $\mathsf{ct}_{w,i_2}^{S_{j,\beta}}$ and samples each honest server's shares of the additive sharing of the ciphertext based on the secret and corrupted servers' shares. While emulating $\mathcal{O}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the output based on the random strings and the output labels. For the same reason in $\mathbf{Hyb}_{14.1.7}$, $\mathbf{Hyb}_{14.\gamma.7}$ and $\mathbf{Hyb}_{14.\gamma.6}$ have the same output distribution.

$\mathbf{Hyb}_{14.\gamma.8}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, for each input wire w of an output gate in $\mathtt{Circ}_\gamma^{V_j}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't follow the protocol to compute honest servers' shared of $\langle\lambda_w\rangle$. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples them based on λ_w and the corrupted servers' shares of $\langle\lambda_w\rangle$. For the same reason in $\mathbf{Hyb}_{14.1.8}$, $\mathbf{Hyb}_{14.\gamma.8}$ and $\mathbf{Hyb}_{14.\gamma.7}$ have the same output distribution.

$\mathbf{Hyb}_{14.\gamma.9}$: In this hybrid, if the receiver R_γ of $[\boldsymbol{s}_\gamma]^{(2)}$ is a virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ and the β-th bit of $\boldsymbol{s}_\gamma$ is used as an input wire with index j_β in $\mathtt{Circ}^{V_j}$, when each honest server $S_{j,i}$ computes $\mathsf{ct}_{j_\beta,1\oplus s_{\gamma,\beta}}^{(\gamma,i)}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ checks whether the query to the random oracle $\mathcal{O}$ has been queried before. If true, $\mathsf{Sim}_{\mathsf{GLOS}}$ aborts the simulation. Otherwise, $\mathsf{Sim}_{\mathsf{GLOS}}$ adds the query to Q. For the same reason in $\mathbf{Hyb}_{14.1.9}$, the distributions of $\mathbf{Hyb}_{14.\gamma.9}$ and $\mathbf{Hyb}_{14.\gamma.8}$ are computationally indistinguishable.

$\mathbf{Hyb}_{14.\gamma.10}$: In this hybrid, if the receiver R_γ of $[\boldsymbol{s}_\gamma]^{(2)}$ is a virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ and the β-th bit of $\boldsymbol{s}_\gamma$ is used as an input wire with index j_β in $\mathtt{Circ}^{V_j}$, for each honest server $S_{j,i}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't follow the protocol to compute $\mathsf{ct}_{j_\beta,1\oplus s_{\gamma,\beta}}^{(\gamma,i)}$. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ samples a random κ-bit string as $\mathsf{ct}_{j_\beta,1\oplus s_{\gamma,\beta}}^{(\gamma,i)}$. While emulating $\mathcal{O}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ computes the output of $\mathcal{O}(\boldsymbol{r}_{1\oplus s_{\gamma,\beta},\beta}\|s_{\gamma,\beta}\|\gamma\|i\|\beta\|j_\beta)$ based on $\mathsf{ct}_{j_\beta,1\oplus s_{\gamma,\beta}}^{(\gamma,i)}$ and $k_{w_{j_\beta},1\oplus v_{w_{j_\beta}}^{S_{j,i}}\oplus\lambda_{w_{j_\beta}}}\|(1\oplus v_{w_{j_\beta}}\oplus\lambda_{w_{j_\beta}})$. For the same reason in $\mathbf{Hyb}_{14.1.10}$, $\mathbf{Hyb}_{14.\gamma.10}$ and $\mathbf{Hyb}_{14.\gamma.9}$ have the same output distribution.

$\mathbf{Hyb}_{14.\gamma.11}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ changes the order of sampling a random κ-bit string as the ciphertext $\mathsf{ct}_{j_\beta,1\oplus s_{\gamma,\beta}}^{(\gamma,i)}$ and sampling $\boldsymbol{r}_{1\oplus s_{\gamma,\beta},\beta,i}$ to decide the queries to $\mathcal{O}$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{14.\gamma.11}$ and $\mathbf{Hyb}_{14.\gamma.10}$ have the same output distribution.

$\mathbf{Hyb}_{14.\gamma.12}$: In this hybrid, for each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$ and each honest server $S_{j,\alpha}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ changes the order of sampling random $c\ell\kappa$-bit strings as the ciphertexts $\mathsf{ct}_{w,i_2}^{S_{j,\beta}}$ for each $\beta = 1,\ldots,c$ and each input wire w of an output gate in $\mathtt{Circ}_\gamma^{V_j}$, and for all $i_2 \in \{0,1\}$ such that $i_2 \neq v_w \oplus \lambda_w$ and sampling $\Delta^{S_{j,\alpha;}}$ to decide the queries to $\mathcal{O}$. Since these two steps are both local computations, this doesn't affect the output distribution. Thus, $\mathbf{Hyb}_{14.\gamma.12}$ and $\mathbf{Hyb}_{14.\gamma.11}$ have the same output distribution.

Note that $\mathbf{Hyb}_{14.\mathsf{rec}.12}$ is just $\mathbf{Hyb}_{14}$, we conclude that the distributions of $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{13}$ are computationally indistinguishable.

Note that for each honest virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, $\Delta^{S_{j,\alpha}}$ is not used before all the ciphertexts of the gates in $\mathtt{Circ}_\gamma^{V_j}$ are generated. $\mathsf{Sim}_{\mathsf{GLOS}}$ delays the generating of $\Delta^{S_{j,\alpha}}$ after the garbling of $\mathtt{Circ}_\gamma^{V_j}$ is completed.

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

$\mathbf{Hyb}_{15}$: In this hybrid, if $\mathsf{Sim}_{\mathsf{GLOS}}$ has emulated $\mathcal{F}_{\mathsf{emulate}}$ to output $\mathsf{success}$, for each AND gate g in each virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$'s local circuit $\mathtt{Circ}^{V_j}$ with input wire a, b and output wire o, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't compute the output of $\mathcal{O}$ to the queries that are used to generate these ciphertexts based on the random strings and the wire labels of wire o. For each output gate in $\mathtt{Circ}^{V_j}$ with input wire w, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't compute the output of $\mathcal{O}$ to the queries that are used to generate these ciphertexts based on the random strings and the output labels. Instead, $\mathsf{Sim}_{\mathsf{GLOS}}$ honestly emulates the random oracles. In particular, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$ are computationally indistinguishable.

For the sake of contradiction, assume that there exists an adversary $\mathcal{A}_1$ such that $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$

are computationally distinguishable. Let Q be the set of queries to the random oracles $\mathcal{O}$ when $\mathsf{Sim}_{\mathsf{GLOS}}$ is computing the ciphertexts for V_j that are randomly generated in the last hybrid. Now we argue that, with non-negligible probability, at least one query in Q has been queried. Suppose this is not the case. Note that all queries in Q are distinct. Then, by assumption, with overwhelming probability, no query in Q has been queried, and all queries in Q are distinct. In this case, the only difference between $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{15}$ is that we do not explicitly compute the output to each query in Q. Since no query in Q has been queried, this makes no difference in the output distribution. Then it shows that $\mathbf{Hyb}_{13}$ and $\mathbf{Hyb}_{12}$ are computationally indistinguishable, which leads to a contradiction.

Thus, with non-negligible probability, at least one query in Q has been queried in $\mathbf{Hyb}_{15}$. However, each query in Q either contains $k_{w,1\oplus v_w\oplus\lambda_w}$ for a wire w in $\mathtt{Circ}^{V_j}$ for some $V_j \in \mathcal{T}_{\mathsf{hon}}$ or contains $\boldsymbol{r}_{1\oplus s_{i,\eta},\eta,\beta}$ for some honest virtual server receiver $R_i = V_j \in \mathcal{T}_{\mathsf{hon}}$ with a honest server $S_{j,\beta}$. Suppose a query contains $k^{S_{j,\alpha}}_{w,1\oplus v_w\oplus\lambda_w}$ for a wire w in an honest virtual server V_j's circuit $\mathtt{Circ}^{V_j}$. Since $\Delta^{S_{j,\beta}}$ 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 $\Delta^{S_{j,\beta}}$. Therefore, $k^{S_{j,\beta}}_{w,1\oplus v_w\oplus\lambda_w} = k^{S_{j,\beta}}_{w,v_w\oplus\lambda_w} \oplus \Delta^{S_{j,\beta}}$ only has $2^{-\kappa+1}\cdot\mathsf{poly}(\kappa)$ probability to be queried by $\mathcal{A}$, which is negligible. Similarly, if a query contains $\boldsymbol{r}_{1\oplus s_{i,\eta},\eta,\beta}$ for some honest virtual server receiver $R_i \in \mathcal{T}_{\mathsf{hon}}$, since $\boldsymbol{r}_{1\oplus s_{i,\eta},\eta,\beta}$ 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 $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$ are computationally indistinguishable.

Note that for virtual server $V_j \in \mathcal{T}_{\mathsf{hon}}$, $\Delta^{S_{j,\beta}}$ is not used in the simulation, $\mathsf{Sim}_{\mathsf{GLOS}}$ 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. $\mathsf{Sim}_{\mathsf{GLOS}}$ 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 $\mathcal{T}_{\mathsf{hon}}$, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't need $\{\boldsymbol{r}^{(\alpha)}_{\mathbf{1}\oplus\boldsymbol{s}_i,\eta}\}^{\kappa}_{\alpha=1}$ in the simulation for honest server $S_{j,\eta}$, so $\mathsf{Sim}_{\mathsf{GLOS}}$ does not generate them in future hybrids. Now all the shares for honest servers with $V_j \in \mathcal{T}_{\mathsf{hon}}$ generated by an honest party in the sharing phase are not used in the simulation, $\mathsf{Sim}_{\mathsf{GLOS}}$ doesn't generate them in future hybrids.

$\mathbf{Hyb}_{16}$: In this hybrid, we change how honest servers compute their outputs. More concretely, $\mathsf{Sim}_{\mathsf{GLOS}}$ sends their outputs to $\mathcal{F}_{\mathsf{GLOS}}$, and honest servers directly obtain their outputs from $\mathcal{F}_{\mathsf{GLOS}}$. This does not change the computational process of honest servers' outputs. Thus, $\mathbf{Hyb}_{16}$ and $\mathbf{Hyb}_{15}$ have the same output distribution.

$\mathbf{Hyb}_{17}$: In this hybrid, when $|\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}| > T$, the preprocessing data for honest virtual servers are not used in the simulation, so $\mathsf{Sim}_{\mathsf{GLOS}}$ does not generate them anymore. Similarly, when $|\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}| \le T$ but the verification of the set does not pass, the preprocessing data for virtual servers out of $\mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{cor}}$ are not used in the simulation, so $\mathsf{Sim}_{\mathsf{GLOS}}$ does not generate them anymore. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{17}$ and $\mathbf{Hyb}_{16}$ have the same output distribution.

Note that $\mathbf{Hyb}_{17}$ is the ideal-world scenario, Π computes $\mathcal{F}_{\mathsf{GLOS}}$ with computational security.

H Security Proof for the Verifiable P2P Channel

We prove Lemma 4 as follows. We prove the security of our verifiable P2P by constructing an ideal adversary Sim. Then we will show that any PPT environment $\mathcal{Z}$ cannot distinguish the output in the ideal world from that in the real world with a non-negligible probability using hybrid arguments.

The ideal adversary Sim is given as follows.

Simulator Sim

To simulate the environment $\mathcal{Z}$, Sim forwards every message it receives from the environment $\mathcal{Z}$ to $\mathcal{A}$ and vice-versa. The simulator simulates the protocol $\Pi_{\mathsf{VP2P}}(A, B)$ as follows:

Send a Message.

- If A and B are both honest, Sim samples a random string (of the same length as the random oracle output) as $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ and sends it to all corrupted parties on behalf of A.
- If A is honest and B is corrupted:
  1. Sim receives m and $(\mathsf{committed}, \mathsf{mid}_m)$ from $\mathcal{F}_{\mathsf{VP2P}}(A, B)$.
  2. Sim samples a random κ-bit string r, honestly emulates $\mathcal{O}$ to compute $\mathcal{O}(m\|r\|\mathsf{mid}_m)$, and broadcasts the result on behalf of A. If $m\|r\|\mathsf{mid}_m$ has been queried by $\mathcal{Z}$, Sim aborts the simulation.
  3. Sim sends (m, r) to B on behalf of A.
  4. If $(\mathsf{complain}, \mathsf{mid}_m, B, A)$ is received from A, Sim sends (m, r) to all the corrupted servers on behalf of A.
  5. Sim sends (m, r) to B on behalf of each honest server.
- If A is corrupted and B is honest:
  1. Sim emulates $\mathcal{O}$ to receive m, r from A and honestly emulates $\mathcal{O}$ to sends the output to A.
  2. Sim receives the broadcast commitment $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ from A and receives (m, r) from A on behalf of B. Sim checks whether $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ matches any of the outputs of $\mathcal{O}$ for a query by $\mathcal{Z}$ of this form. If there is a unique pair of (m, r) such that $m\|r\|\mathsf{mid}_m$ has been queried by $\mathcal{Z}$ and matches $\mathcal{O}(m\|r\|\mathsf{mid}_m)$, then Sim sends the corresponding m to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. If at least two pairs of (m, r) matches $\mathcal{O}(m\|r\|\mathsf{mid}_m)$, Sim aborts the simulation. If no such pair of (m, r), Sim sends $\perp$ to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$.
  3. Sim honestly emulates $\mathcal{O}$ and checks the commitment on behalf of B. If the check fails, Sim broadcasts $(\mathsf{complain}, \mathsf{mid}_m, B, A)$ on behalf of B. If the check passes, but the message m doesn't match what Sim sends to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$, Sim aborts the simulation.
  4. If $(\mathsf{complain}, \mathsf{mid}_m, B, A)$ has been broadcast, Sim receives (m, r) from A on behalf of each honest server. Sim honestly emulates $\mathcal{O}$ and checks the commitment on behalf of the honest server. If the check fails, Sim broadcasts $(\mathsf{eliminate}, A)$ on behalf of the honest server. If the check passes, but the message m doesn't match what Sim sends to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$, Sim aborts the simulation.
  5. If $\geq t + 1$ servers broadcast $(\mathsf{eliminate}, A)$ (including the broadcasts by honest servers which are simulated by Sim and the broadcasts by corrupted servers), then Sim sends $(\mathsf{eliminate}, A)$ to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. Otherwise, Sim sends $(\mathsf{deliver}, \mathsf{mid}_m)$ to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$.
- If A and B are both corrupted:
  1. Sim emulates $\mathcal{O}$ to receive m, r from A and honestly emulates $\mathcal{O}$ to sends the output to A.
  2. Sim receives the broadcast commitment $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ from A. Sim checks whether $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ matches any of the outputs of $\mathcal{O}$ for a query by $\mathcal{Z}$ of this form. If there is a unique pair of (m, r) such that $m\|r\|\mathsf{mid}_m$ has been queried by $\mathcal{Z}$ and matches $\mathcal{O}(m\|r\|\mathsf{mid}_m)$, then Sim sends the corresponding m to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. If at least two pairs of (m, r) matches $\mathcal{O}(m\|r\|\mathsf{mid}_m)$, Sim aborts the simulation. If no such pair of (m, r), Sim sends $\perp$ to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$.
  3. If $(\mathsf{complain}, \mathsf{mid}_m, B, A)$ has been broadcast, Sim receives (m, r) from A on behalf of each honest server. Sim honestly emulates $\mathcal{O}$ and checks the commitment on behalf of the honest server. If the check fails, Sim broadcasts $(\mathsf{eliminate}, A)$ on behalf of the honest server. If the check passes, but the message m doesn't match what Sim sends to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$, Sim aborts the simulation.
  4. If $\geq t + 1$ servers broadcast $(\mathsf{eliminate}, A)$ (including the broadcasts by honest servers which are simulated by Sim and the broadcasts by corrupted servers), then Sim sends $(\mathsf{eliminate}, A)$ to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. Otherwise, Sim sends $(\mathsf{deliver}, \mathsf{mid}_m)$ to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$.

Open a Message. When A (resp. B) is requested to open a message m with message index mid_m:

- If A (resp. B) is honest:
  1. Sim receives m from $\mathcal{F}_{\mathsf{VP2P}}(A, B)$.
  2. If A, B are both honest, Sim randomly samples a κ-bit string as r and sends (m, r) to all the corrupted servers on behalf of A (resp. B). If $m\|r\|\mathsf{mid}_m$ has been queried by $\mathcal{Z}$, Sim aborts the simulation. If B (resp. A) is corrupted, Sim follows the protocol to send (m, r) to all the corrupted servers on behalf of A (resp. B).
- If A (resp. B) is corrupted:

1. Sim initializes $S = \emptyset$.
2. Sim receives (m, r) from A (resp. B) on behalf of each honest server. Sim honestly emulates $\mathcal{O}$ and checks the commitment on behalf of the honest server. If the check fails, Sim broadcasts $(\mathsf{eliminate}, A)$ (resp. $(\mathsf{eliminate}, B)$) on behalf of the honest server. If the check passes, Sim adds this honest server into S, and if the received m doesn't match the message Sim sends to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$ when sending the message with index mid_m, Sim aborts the simulation.
3. If $\geq t + 1$ servers broadcast $(\mathsf{eliminate}, A)$ (resp. $(\mathsf{eliminate}, B)$), then Sim sends $(\mathsf{eliminate}, A)$ (resp. $(\mathsf{eliminate}, B)$) to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. Otherwise, Sim sends $(\mathsf{open}, \mathsf{mid}_m, S)$ to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$.

Figure 23: The simulator for the verifiable P2P channel.

We construct the following hybrids:

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

Hyb$_1$: In this hybrid, during **Send a Message**, if A, B are both honest, Sim does not emulate the random oracle honestly and broadcasts the output $\mathcal{O}(m\|r\|\mathsf{mid}_m)$. Instead, Sim directly samples a random string as $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ and delays the generation of r until m is requested to be opened. During **Open a Message**, when m is requested to be opened, if $m\|r\|\mathsf{mid}_m$ has been queried by $\mathcal{Z}$, Sim aborts the simulation. Note that if $\mathcal{Z}$ has not queried $m\|r\|\mathsf{mid}_m$ before m is opened, then the RO output $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ is uniformly random, so we do not change anything in this case except that we delay the generation of r until it is needed. This does not affect the output distribution. Therefore, we only change the output distribution when $\mathcal{Z}$ queries $m\|r\|\mathsf{mid}_m$ before m is opened, i.e., before r is sampled. Since r is a uniformly random κ-bit string, and $\mathcal{Z}$ only has $\mathsf{poly}(\kappa)$ queries to the random oracle, this only happens with $\mathsf{poly}(\kappa)/2^\kappa$ probability, which is negligible. Thus, the distributions of **Hyb**$_1$ and **Hyb**$_0$ are computationally indistinguishable.

Hyb$_2$: In this hybrid, during **Send a Message**, if A, B are both honest, Sim does not receive the message m from the environment $\mathcal{Z}$. Instead, Sim receives it from $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. Since the honest party's message m is exactly the output from $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. The message m from $\mathcal{Z}$ and from $\mathcal{F}_{\mathsf{VP2P}}(A, B)$ are the same. Thus, **Hyb**$_2$ and **Hyb**$_1$ have the same distribution.

Hyb$_3$: In this hybrid, during **Send a Message**, if A is honest and B is corrupted, after sampling r, if $m\|r\|\mathsf{mid}_m$ has been queried by $\mathcal{Z}$ before, Sim aborts the simulation. Since r is a uniformly random κ-bit string, and $\mathcal{Z}$ only has $\mathsf{poly}(\kappa)$ queries to the random oracle, this only happens with $\mathsf{poly}(\kappa)/2^\kappa$ probability, which is negligible. Thus, the distributions of **Hyb**$_3$ and **Hyb**$_2$ are computationally indistinguishable.

Hyb$_4$: In this hybrid, during **Send a Message** and **Open a Message**, if A is honest and B is corrupted, Sim does not receive the message m from the environment $\mathcal{Z}$. Instead, Sim receives it from $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. Since the honest party's message m sent to B is exactly the output from $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. The message m from $\mathcal{Z}$ and from $\mathcal{F}_{\mathsf{VP2P}}(A, B)$ are the same. Thus, **Hyb**$_4$ and **Hyb**$_3$ have the same distribution.

Hyb$_5$: In this hybrid, during **Send a Message**, if A is corrupted, after Sim receives the broadcast commitment $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ from A, Sim checks whether $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ matches any of the outputs of $\mathcal{O}$ for a query by $\mathcal{Z}$ of this form. If there is a unique pair of (m, r) such that $m\|r\|\mathsf{mid}_m$ has been queried by $\mathcal{Z}$ and matches $\mathcal{O}(m\|r\|\mathsf{mid}_m)$, then Sim sends the corresponding m to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. If at least two pairs of (m, r) matches $\mathcal{O}(m\|r\|\mathsf{mid}_m)$, Sim aborts the simulation. If no such pair of (m, r), Sim sends $\perp$ to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. Since the outputs from $\mathcal{F}_{\mathsf{VP2P}}(A, B)$ are not used in the simulation, sending an input to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$ does not affect the output distribution. Therefore, the output distribution only changes when $\mathcal{Z}$ queries the random oracle with $m_1\|r_1\|\mathsf{mid}_m, m_2\|r_2\|\mathsf{mid}_m$ with $\mathcal{O}(m_1\|r_1\|\mathsf{mid}_m) = \mathcal{O}(m_2\|r_2\|\mathsf{mid}_m)$. Note that the output of a query to $\mathcal{O}$ is uniformly random and the output length of $\mathcal{O}$ is larger than κ, there is only $\mathsf{poly}(\kappa)/2^\kappa$ probability that two different queries by $\mathcal{Z}$ have the same outputs. Therefore, the distribution only changes with a negligible probability. Thus, the distributions of **Hyb**$_5$ and **Hyb**$_4$ are computationally indistinguishable.

Hyb$_6$: In this hybrid, during **Send a Message**, if A is corrupted and B is honest, after checking the commitment of m from A on behalf of B, Sim aborts the simulation if the check passes but the message m sent by A does not match the message sent to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. This happens only when $\mathcal{Z}$ finds a pair of $(m', r') \neq (m, r)$ such that $\mathcal{O}(m'\|r'\|\mathsf{mid}_m) = \mathcal{O}(m\|r\|\mathsf{mid}_m)$, or for a string s of the same length as a random

oracle output, $\mathcal{Z}$ finds a pair of (m, r) such that $\mathcal{O}(m\|r\|\mathsf{mid}_m) = s$. As we argued in the last hybrid, the first case happens only with a negligible probability. Since $\mathcal{O}(m\|r\|\mathsf{mid}_m)$ is a uniformly random string before $\mathcal{Z}$ queries the RO with $m\|r\|\mathsf{mid}_m$, the probability that each RO output matches s is at most $2^{-\kappa}$. Taking the union bound for all $\mathsf{poly}(\kappa)$ queries, the second case happens only with $\mathsf{poly}(\kappa)/2^{\kappa}$ probability, which is also negligible. Thus, the distributions of $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ are computationally indistinguishable.

$\mathbf{Hyb}_7$: In this hybrid, during **Send a Message** and **Open a Message**, if A is corrupted, and B has broadcast $(\mathsf{complain}, \mathsf{mid}_m, B, A)$, after checking the commitment of m from A on behalf of each honest server, Sim aborts the simulation if the check passes but the message m sent by A does not match the message sent to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. For the same reason as in the last hybrid, the output 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, during **Send a Message**, if A is corrupted, honest parties do not compute their output by themselves. Instead, if $\geq t+1$ servers broadcast $(\mathsf{eliminate}, A)$ (including the broadcasts by honest servers which are simulated by Sim and the broadcasts by corrupted servers), then Sim sends $(\mathsf{eliminate}, A)$ to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. Otherwise, Sim sends $(\mathsf{deliver}, \mathsf{mid}_m)$ to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. Then, honest parties get their outputs from $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. When there are $\geq t + 1$ servers that broadcast $(\mathsf{eliminate}, A)$, honest parties will get $(\mathsf{eliminate}, A)$ as their outputs, which is the same as the output sent by $\mathcal{F}_{\mathsf{VP2P}}(A, B)$ in this case. On the other hand, when only $< t + 1$ servers broadcast $(\mathsf{eliminate}, A)$, at least one honest server gets the message m sent to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$ and passes the check. That is to say, this honest server will deliver m to B, and B will accept m if he is honest. Therefore, the output distribution also remains unchanged. Thus, $\mathbf{Hyb}_8$ and $\mathbf{Hyb}_7$ have the same distribution.

$\mathbf{Hyb}_9$: In this hybrid, during **Open a Message**, if A (resp. B) is corrupted, Sim initializes a set $S = \emptyset$ at the beginning. When an honest server's check on the opened message passes, Sim adds the honest server into S. This does not affect the output distribution. Thus, $\mathbf{Hyb}_9$ and $\mathbf{Hyb}_8$ have the same distribution.

$\mathbf{Hyb}_{10}$: In this hybrid, during **Open a Message**, if A (resp. B) is corrupted, honest parties do not compute their output by themselves. Instead, if $\geq t+1$ servers broadcast $(\mathsf{eliminate}, A)$ (resp. $(\mathsf{eliminate}, B)$), then Sim sends $(\mathsf{eliminate}, A)$ (resp. $(\mathsf{eliminate}, B)$) to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. Otherwise, Sim sends the request $(\mathsf{open}, \mathsf{mid}_m, S)$ to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. Then, honest parties get their outputs from $\mathcal{F}_{\mathsf{VP2P}}(A, B)$. When there are $\geq t + 1$ servers broadcasting $(\mathsf{eliminate}, A)$, honest parties will get $(\mathsf{eliminate}, A)$ as their outputs, which is the same as the output sent by $\mathcal{F}_{\mathsf{VP2P}}(A, B)$ in this case. On the other hand, when only $< t + 1$ servers broadcast $(\mathsf{eliminate}, A)$, the parties in S will accept the message m sent by Sim to $\mathcal{F}_{\mathsf{VP2P}}(A, B)$, and the output distribution also remains unchanged. Thus, $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ have the same distribution.

Note that $\mathbf{Hyb}_{10}$ is the ideal-world scenario, so the protocol $\Pi_{\mathsf{VP2P}}(A, B)$ realizes $\mathcal{F}_{\mathsf{VP2P}}(A, B)$ with compuational security.

I Detailed Protocol Description of Π_{CompVer}

The protocol Π_{CompVer} for checking the servers' local computations is given as follows.

Protocol Π_{CompVer}

1. The servers invoke $\mathcal{F}_{\mathsf{Coin}}$ with $\mathsf{RandCoin}$ to receive $r_{\mathsf{ver}} \in \mathbb{F}_{2^\kappa}$. Then, the parties use $\mathcal{O}(r_{\mathsf{ver}})$ to decide a subset $\mathcal{T}_{\mathsf{ver}}$ of:
   - $N/240$ virtual servers if the protocol has not been rerun (i.e. $\mathsf{Rerun} = 0$ in the main protocol Π_{Main}).
   - $N'/240$ virtual servers otherwise.
2. For each virtual server $V_j \in \mathcal{T}_{\mathsf{ver}}$, and each message m (with message index mid_m) sent from $S_{j,\alpha}$ to $S_{j,\beta}$ during the execution of Π_{Gb}, all the parties send $(\mathsf{open}_{S_{j,\alpha}}, \mathsf{mid}_m), (\mathsf{open}_{S_{j,\beta}}, \mathsf{mid}_m)$ to $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$. If $(\mathsf{eliminate}, S_{j,\alpha})$ (resp. $(\mathsf{eliminate}, S_{j,\beta})$) is received, the parties broadcast $(\mathsf{eliminate}, S_{j,\alpha})$ (resp. $(\mathsf{eliminate}, S_{j,\beta})$). If $(\mathsf{eliminate}, S_{j,\alpha})$ (resp. $(\mathsf{eliminate}, S_{j,\beta})$) is received from $t + 1$ servers, the parties mark $S_{j,\alpha}$ (resp. $S_{j,\beta}$) "eliminated".
3. For each $V_j \in \mathcal{T}_{\mathsf{ver}}$, for each message m with message index mid_m in Π_{Share} sent from a party P to a member $S_{j,\alpha}$ of V_j, all the servers send $(\mathsf{open}_{S_{j,\alpha}}, \mathsf{mid}_m)$ to $\mathcal{F}_{\mathsf{VP2P}}(P, S_{j,\alpha})$. If $(\mathsf{eliminate}, S_{j,\alpha})$ is received, the servers

broadcast $(\mathsf{eliminate}, S_{j,\alpha})$. If $(\mathsf{eliminate}, S_{j,\alpha})$ is received from $t+1$ servers, the parties mark $S_{j,\alpha}$ (resp. $S_{j,\beta}$) "eliminated".

4. For each $V_j \in \mathcal{T}_{\mathsf{ver}}$, for the ROT correlations sent from $\mathcal{F}_{\mathsf{BaseROT}}$ to two members $(S_{j,\alpha}, S_{j,\beta})$ of V_j, all the servers hold the VSSs of them. Then, all the parties send their shares to each other and then reconstruct the base ROTs for $(S_{j,\alpha}, S_{j,\beta})$.
5. For each $V_j \in \mathcal{T}_{\mathsf{ver}}$, each server $S_{j,\alpha}$ $(\alpha = 1, \ldots, c)$ sends his output $\mathsf{gbout}_{j,\alpha}$ from $\Pi_{\mathsf{Gb}}(V_j)$ together with the randomness $r_{j,\alpha}$ (used for the commitment) to all the parties. Then, each server checks whether $\mathsf{gbout}_{j,\alpha}, r_{j,\alpha}$ matches the commitment $\mathcal{O}(\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$. If not, the party broadcasts $(\mathsf{eliminate}, S_{j,\alpha})$.
6. For each $V_j \in \mathcal{T}_{\mathsf{ver}}$, each server $S_{j,\alpha}$ $(\alpha = 1, \ldots, c)$ sends his seed $\mathsf{seed}_{j,\alpha}$ for generating the randomness in his execution of Π_{Gb} to all the servers together with the random string $s_{j,\alpha}$ for committing to this seed. Then, each server checks whether $\mathsf{seed}_{j,\alpha}, s_{j,\alpha}$ matches the commitment $\mathcal{O}(\mathsf{seed}_{j,\alpha} \| s_{j,\alpha} \| j \| \alpha)$. If not, the party broadcasts $(\mathsf{eliminate}, S_{j,\alpha})$.
7. Each server now obtains the view of each server $S_{j,\alpha}$ in $\{\Pi_{\mathsf{Gb}}(V_j)\}_{V_j \in \mathcal{T}_{\mathsf{ver}}}$ and Π_{ShVer}. Then, each server checks whether $S_{j,\alpha}$ behaves honestly during the protocol executions. If not, the server broadcasts $(\mathsf{eliminate}, S_{j,\alpha})$.
8. If $(\mathsf{eliminate}, S_{j,\alpha})$ is received from at least $t+1$ servers, the parties mark $S_{j,\alpha}$ "eliminated".

Figure 24: The verification protocol for the local computations.

J The Security Proof for the Main Protocol

We prove the security of our main protocol by constructing an ideal adversary Sim. Then we will show that the output in the ideal world is indistinguishable from that in the real world against the PPT environment $\mathcal{Z}$ using hybrid arguments. Our proof relies on the simulator $\mathsf{Sim}_{\mathsf{GLOS}}$ constructed in Section G.

We construct the ideal adversary Sim for our main protocol as follows.

Simulator Sim

To simulate the environment $\mathcal{Z}$, Sim forwards every message it receives from the environment $\mathcal{Z}$ to $\mathcal{A}$ and vice-versa. Sim follows the protocol to initialize and compute $\mathsf{Rerun}, \mathcal{P}_S$ during the whole simulation. Sim reruns the simulation from Step 2 and sets $\mathsf{Rerun} = 1$ once $\mathcal{P}_S < n - t/2$ and $\mathsf{Rerun} = 0$. The virtual servers emulated completely by servers in $\mathcal{P}_S$ are called active ones.

1. **VSS Inputs.** For each input wire w with input x attached to a client C_i, if C_i is corrupted, Sim emulates $\mathcal{F}_{\mathsf{VSS}}$ to receive x and the randomness r for generating $[\![x]\!]$ from $\mathcal{A}$. If C_i is honest, Sim receives the corrupted servers' shares of $[\![x]\!]$. Then, Sim faithfully emulates $\mathcal{F}_{\mathsf{VSS}}$ to send the corrupted servers' shares to them. Sim now obtains all the corrupted clients' inputs, and then Sim sends them to $\mathcal{F}$ to get the corrupted clients' outputs. Sim then initializes the corresponding protocol Π modified from the protocol Π_{GLOS} of [GLOS25] (constructed in Section G) with the determined parameter.
2. **Determining Virtual Servers.** Sim runs $\mathsf{Sim}_{\mathsf{GLOS}}$ to obtain r_{vp}. Then, Sim emulates $\mathcal{F}_{\mathsf{Coin}}$ to send it to all the parties and follows the protocol to determine the virtual servers. In addition, Sim samples a random set of $N/240$ virtual servers as $\mathcal{T}_{\mathsf{ver}}$. Then, Sim emulates $\mathcal{F}_{\mathsf{GLOS}}$ to send $\mathcal{T}_{\mathsf{ver}}$ to $\mathsf{Sim}_{\mathsf{GLOS}}$, where $\mathsf{Sim}_{\mathsf{GLOS}}$ is simulator constructed in Section G that interacts with an environment $\mathcal{Z}'$ (which includes the adversary $\mathcal{A}'$ for the protocol Π constructed in Section G) and the functionality $\mathcal{F}_{\mathsf{GLOS}}$. Both $\mathcal{Z}'$ and $\mathcal{F}_{\mathsf{GLOS}}$ are emulated by Sim here, where the messages sent by $\mathcal{F}_{\mathsf{GLOS}}$ and $\mathcal{A}'$ are determined later. In particular, when $\mathsf{Sim}_{\mathsf{GLOS}}$ invokes $\mathcal{F}_{\mathsf{GLOS}}$, Sim emulates $\mathcal{F}_{\mathsf{GLOS}}$ to send corrupted clients' outputs (which are obtained in Step 1). Additionally, Sim initializes $\mathsf{Comp}_j = 0$ for each virtual server V_j in this execution of Π_{GLOS}. Let $\mathcal{T}_{\mathsf{cor}}$ be the set of corrupted virtual servers. If $|\mathcal{T}_{\mathsf{cor}}| > N/60$ and $\mathsf{Rerun} = 0$, Sim aborts the simulation. If $|\mathcal{T}_{\mathsf{cor}}| > N'/120$ and $\mathsf{Rerun} = 1$, Sim also aborts the simulation.
3. **Generating Sharings.** Sim runs $\mathsf{Sim}_{\mathsf{Share}}$ (Figure 26).
4. **Generating Auxiliary Sharings.** Let $\kappa' = N + \kappa$. Then, for each party P:

- If P is honest:
  (a) For each share with message index mid_m sent to a server $S_{j,\alpha}$ in this step, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(P, S_{j,\alpha})$ to send $(\mathsf{committed}, \mathsf{mid}_m)$ to each corrupted party.
  (b) Sim randomly samples the shares of $[\boldsymbol{r}_1^P]_{\times\kappa'}, [\boldsymbol{r}_2^P]^{(2)}_{\times\kappa'}, [\boldsymbol{r}_3^P]^{(3)}_{\times\kappa'}, [\boldsymbol{o}^P]^{(2)}_{\times\kappa'}$ for the corrupted virtual servers. Then, for each V_j among them, Sim randomly samples the additive sharings for V_j's shares based on the secrets. For each other virtual server V_j, Sim randomly samples the shares sent to each corrupted server $S_{j,\alpha}$.
  (c) Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(P, S_{j,\alpha})$ to send each corrupted server $S_{j,\alpha}$'s shares to $S_{j,\alpha}$.
- If P is corrupted, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(P, S_{j,\alpha})$ to receive each message m with index mid_m to $S_{j,\alpha}$ from P. Then, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(P, S_{j,\alpha})$ to receive a request from $\mathcal{A}$:
  * If the request is $(\mathsf{deliver}, \mathsf{mid}_m)$, then if $S_{j,\alpha}$ is honest, Sim sends m to $S_{j,\alpha}$. Sim then emulates $\mathcal{F}_{\mathsf{VP2P}}(P, S_{j,\alpha})$ to send $(\mathsf{committed}, \mathsf{mid}_m)$ to each corrupted party.
  * Otherwise, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(P, S_{j,\alpha})$ to send $(\mathsf{eliminate}, P)$ to all the corrupted parties. If P is a corrupted server, Sim eliminates P from $\mathcal{P}_S$.

5. **Preparing Base ROTs.** Sim emulates $\mathcal{F}_{\mathsf{BaseROT}}$ to receive ROT from the corrupted servers and receive corrupted servers' outputs of $\mathcal{F}_{\mathsf{BaseROT}}$ from $\mathcal{A}$. Sim then extracts corrupted servers' outputs of $\mathcal{F}_{\mathsf{GLOSbaseROT}}$ from the received outputs. Then, Sim sets their outputs of $\mathcal{F}_{\mathsf{GLOSbaseROT}}$ in Π to be the same messages while emulating $\mathcal{A}'$ to interact with $\mathsf{Sim}_{\mathsf{GLOS}}$.
6. **Verifying the Sharings.** Sim runs $\mathsf{Sim}_{\mathsf{ShVer}}$ (Figure 27).
7. **Committing the Randomness.** For each server $S_{j,\alpha}$ who emulates a virtual server V_j:
   - If $S_{j,\alpha}$ is honest, Sim samples a random string (with the same length as an RO output) as $\mathcal{O}(\mathsf{seed}_{j,\alpha} \| s_{j,\alpha} \| j \| \alpha)$ and broadcasts it on behalf of $S_{j,\alpha}$.
   - If $S_{j,\alpha}$ is corrupted, Sim emulates $\mathcal{O}$ to receive $\mathsf{seed} \| s \| j \| \alpha$ from $\mathcal{A}$ and honestly emulates $\mathcal{O}$ to send the output to $\mathcal{A}$. Sim then receives $S_{j,\alpha}$'s broadcast $\mathcal{O}(\mathsf{seed}_{j,\alpha} \| s_{j,\alpha} \| j \| \alpha)$ and checks whether it matches an output of $\mathcal{O}$ with a query of the form $\mathsf{seed} \| s \| j \| \alpha$. If there are two such pairs of (seed, s), Sim aborts the simulation. If there is a single pair of (seed, s) such that $\mathcal{O}(\mathsf{seed}_{j,\alpha} \| s_{j,\alpha} \| j \| \alpha)$ matches $\mathcal{O}(\mathsf{seed} \| s \| j \| \alpha)$, Sim sets $(\mathsf{seed}_{j,\alpha}, s_{j,\alpha})$ to be (seed, s). If no such a pair of (seed, s), Sim sets $(\mathsf{seed}_{j,\alpha}, s_{j,\alpha}) = 0$.

   Now, Sim has obtained each corrupted server $S_{j,\alpha}$'s input set $I_{j,\alpha}$ to $\Pi_{\mathsf{Gb}}(V_j)$. Sim then sets the message sent by $S_{j,\alpha}$ in Step 5 of Π to $(\mathsf{query}, S_{j,\alpha}, I_{j,\alpha} \| s_{j,\alpha})$ while emulating $\mathcal{A}'$ to interact with $\mathsf{Sim}_{\mathsf{GLOS}}$.
8. **Garbling the Computations.**
   (a) Sim runs $\mathsf{Sim}_{\mathsf{Gb}}$ (Figure 28) to simulate the execution of $\Pi_{\mathsf{Gb}}(V_j)$ for each active V_j.
   (b) For each honest server $S_{j,\alpha}$ emulating an active virtual server V_j, Sim samples a random string with the same length as an output of $\mathcal{O}$ as $\mathcal{O}(\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$ and sends it to all the corrupted parties as $S_{j,\alpha}$'s broadcast.
   (c) For each corrupted server $S_{j,\alpha}$, Sim honestly emulates $\mathcal{O}$ to receive queries of the form $\mathsf{gbout} \| r \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ and send the output to $\mathcal{A}$. Then, Sim receives $\mathcal{O}(\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$ from $S_{j,\alpha}$'s broadcast. Sim checks whether there is a unique gbout such that $\mathcal{O}(\mathsf{gbout} \| r \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}) = \mathcal{O}(\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$. If true, $(\mathsf{gbout}_{j,\alpha}, r_{j,\alpha})$ is set to (gbout, r). If there are two such pairs of $(\mathsf{gbout}_{j,\alpha}, r_{j,\alpha})$, Sim aborts the simulation. If no such a pair of $(\mathsf{gbout}_{j,\alpha}, r_{j,\alpha})$, Sim sets $(\mathsf{gbout}_{j,\alpha}, r_{j,\alpha})$ to be 0.
9. **Verifying the Computations.** Sim runs $\mathsf{Sim}_{\mathsf{CompVer}}$ (Figure 29).
10. **Collecting Garbled Circuits.** Sim runs $\mathsf{Sim}_{\mathsf{Collect}}$ (Figure 30).
11. **Evaluating Garbled Circuits.** No communication needs to be simulated in this step.

Figure 25: The simulator for our main protocol Π_{Main}.

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

For each message with message index mid_m sent from a server S_i to another server S_j during Π_{Share}, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to receive $(\mathsf{send}_{S_i,S_j}, \mathsf{mid}_m)$ from the corrupted parties. Then:

- If S_i, S_j are both honest, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to send $(\mathsf{committed}, \mathsf{mid}_m)$ to each corrupted party.
- If S_i is honest and S_j is corrupted, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to send $(\mathsf{committed}, \mathsf{mid}_m)$ to each corrupted party. In addition, Sim runs $\mathsf{Sim}_{\mathsf{GLOS}}$ to generate the message m and emulates $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to send it to S_j.
- If S_j is honest and S_i is corrupted, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to receive the message m from S_i. Then, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to receive the request from $\mathcal{A}$.
  - If the request is $(\mathsf{deliver}, \mathsf{mid}_m)$, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to send $(\mathsf{committed}, \mathsf{mid}_m)$ to each corrupted party.
  - Otherwise, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to send $(\mathsf{eliminated}, S_i)$ to each corrupted party and follows the protocol to update $\mathcal{P}_S$. All the messages sent from S_i during the protocol Π constructed in Section G are set to 0 while emulating $\mathcal{A}'$ to interact with $\mathsf{Sim}_{\mathsf{GLOS}}$, and Sim does not simulate the messages sent to S_i from the next round of the protocol (same below for each server eliminated from $\mathcal{P}_S$).
- If S_i, S_j are both corrupted, $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to receive the message m from S_i. Then, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to receive the request from $\mathcal{A}$.
  - If the request is $(\mathsf{deliver}, \mathsf{mid}_m)$, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to send $(\mathsf{committed}, \mathsf{mid}_m)$ to each corrupted party.
  - Otherwise, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$ to send $(\mathsf{eliminated}, S_i)$ to each corrupted party and follows the protocol to update $\mathcal{P}_S$. All the messages sent from S_i during the protocol Π constructed in Section G are set to 0 while emulating $\mathcal{A}'$ to communicate with $\mathsf{Sim}_{\mathsf{GLOS}}$.

Figure 26: The simulator for the step of generating sharings.

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

1. Sim emulates $\mathcal{F}_{\mathsf{Coin}}$ to receive $\mathsf{RandomCoin}$ from the corrupted parties and faithfully emulates $\mathcal{F}_{\mathsf{Coin}}$ to sample $s_{\mathsf{ver}} \in \mathbb{F}_{2^\kappa}$ and follows the protocol to emulate $\mathcal{O}$ and expands it. Then, Sim emulates $\mathcal{F}_{\mathsf{Coin}}$ to send s_{ver} to all the parties. If s_{ver} has been queried by $\mathcal{A}$, Sim aborts the simulation.
2. For each honest party P:
   (a) For each corrupted server $S_{j,\alpha}$, Sim follows the protocol to compute $S_{j,\alpha}$'s shares of $\langle([\tau_1^P]_{\kappa'})^{V_j}\rangle, \langle([\tau_2^P]_{\kappa'}^{(2)})^{V_j}\rangle, \langle([\tau_3^P]_{\kappa'}^{(3)})^{V_j}\rangle, \langle([\tau_4^P]_{\kappa'}^{(2)})^{V_j}\rangle$, where $([\tau_1^P]_{\kappa'})^{V_j}$ denotes V_j's share of $[\tau_1^P]_{\kappa'}$ and similar for the other three shares.
   (b) Sim reconstructs the shares of $[\tau_1^P]_{\kappa'}, [\tau_2^P]_{\kappa'}^{(2)}, [\tau_3^P]_{\kappa'}^{(3)}, [\tau_4^P]_{\kappa'}^{(2)}$ for each corrupted virtual server V_j. Sim samples the whole sharings $[\tau_1^P]_{\kappa'}, [\tau_2^P]_{\kappa'}^{(2)}, [\tau_3^P]_{\kappa'}^{(3)}, [\tau_4^P]_{\kappa'}^{(2)}$ based on these shares.
   (c) For each other virtual server V_j, Sim samples the additive sharings $\langle([\tau_1^P]_{\kappa'})^{V_j}\rangle, \langle([\tau_2^P]_{\kappa'}^{(2)})^{V_j}\rangle, \langle([\tau_3^P]_{\kappa'}^{(3)})^{V_j}\rangle, \langle([\tau_4^P]_{\kappa'}^{(2)})^{V_j}\rangle$, where $([\tau_1^P]_{\kappa'})^{V_j}$ based on the corrupted servers' shares and the secrets.
   (d) Finally, Sim broadcasts $[\tau_1^P]_{\kappa'}, [\tau_2^P]_{\kappa'}^{(2)}, [\tau_3^P]_{\kappa'}^{(3)}, [\tau_4^P]_{\kappa'}^{(2)}$ on behalf of P and broadcasts each honest server $S_{j,\alpha}$'s shares of $\langle([\tau_1^P]_{\kappa'})^{V_j}\rangle, \langle([\tau_2^P]_{\kappa'}^{(2)})^{V_j}\rangle, \langle([\tau_3^P]_{\kappa'}^{(3)})^{V_j}\rangle, \langle([\tau_4^P]_{\kappa'}^{(2)})^{V_j}\rangle$ on behalf of $S_{j,\alpha}$. Then, Sim receives each corrupted server $S_{j,\alpha}$'s shares of $\langle([\tau_1^P]_{\kappa'})^{V_j}\rangle, \langle([\tau_2^P]_{\kappa'}^{(2)})^{V_j}\rangle, \langle([\tau_3^P]_{\kappa'}^{(3)})^{V_j}\rangle, \langle([\tau_4^P]_{\kappa'}^{(2)})^{V_j}\rangle$ from $S_{j,\alpha}$ and checks whether they are correctly sent.
3. For each corrupted party P, Sim follows the protocol to compute each honest server $S_{j,\alpha}$'s shares of $\langle([\tau_1^P]_{\kappa'})^{V_j}\rangle, \langle([\tau_2^P]_{\kappa'}^{(2)})^{V_j}\rangle, \langle([\tau_3^P]_{\kappa'}^{(3)})^{V_j}\rangle, \langle([\tau_4^P]_{\kappa'}^{(2)})^{V_j}\rangle$ and broadcast them on behalf of $S_{j,\alpha}$. Sim then receives P's broadcasts of $[\tau_1^P]_{\kappa'}, [\tau_2^P]_{\kappa'}^{(2)}, [\tau_3^P]_{\kappa'}^{(3)}, [\tau_4^P]_{\kappa'}^{(2)}$ on behalf of each honest party.
4. For each party P, Sim follows the protocol to compute each corrupted server $S_{j,\alpha}$'s shares of $\langle([\tau_1^P]_{\kappa'})^{V_j}\rangle, \langle([\tau_2^P]_{\kappa'}^{(2)})^{V_j}\rangle, \langle([\tau_3^P]_{\kappa'}^{(3)})^{V_j}\rangle, \langle([\tau_4^P]_{\kappa'}^{(2)})^{V_j}\rangle$ from the committed messages by $\mathcal{F}_{\mathsf{VP2P}}(S, S_{j,\alpha})$ and checks whether they match the broadcast from $S_{j,\alpha}$. If not, Sim sets $\mathsf{Comp}_j = 1$.
5. Sim follows the protocol to check whether the sharings $[\tau_1^P]_{\kappa'}, [\tau_2^P]_{\kappa'}^{(2)}, [\tau_3^P]_{\kappa'}^{(3)}, [\tau_4^P]_{\kappa'}^{(2)}$ are all valid for each corrupted P. If not, Sim marks P "$\mathsf{eliminated}$" and follows the protocol to update $\mathcal{P}_S$. All the messages sent from P during the protocol Π are set to 0 while emulating $\mathcal{A}'$ to interact with $\mathsf{Sim}_{\mathsf{GLOS}}$.

6. Sim follows the protocol to check whether the broadcasts from each pair of parties (P_i, P_j) do not match (for the linear combinations of shares sent from P_i to P_j). If not, let the message of the shares sent from P_i to P_j in Π_{Share} be mid_m, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(P_i, P_j)$ to receive $(\mathsf{open}_{P_i}, \mathsf{mid}_m)$ from the corrupted parties. Then:
   (a) If P_i is honest, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(P_i, P_j)$ to send the message m with index mid_m to all the corrupted parties.
   (b) If S_i is corrupted, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(P_i, P_j)$ to receive a request from $\mathcal{A}$:
      – If $(\mathsf{open}, \mathsf{mid}_m, \mathsf{Set})$ is received, Sim broadcasts $(\mathsf{eliminate}, P_i)$ on behalf of each honest party out of Set. Then, if the message m with index mid_m does not match the broadcast from P_i, Sim broadcasts $(\mathsf{eliminate}, P_i)$ on behalf of each honest party in Set. If there are at least $t+1$ parties who broadcast $(\mathsf{eliminate}, P_i)$ (including the broadcasts received from corrupted parties and the broadcasts of honest parties emulated by Sim), Sim marks P_i "$\mathsf{eliminated}$" and follows the protocol to update $\mathcal{P}_S$. All the messages sent from P_i during Π are set to 0 while emulating $\mathcal{A}'$ to interact with $\mathsf{Sim}_{\mathsf{GLOS}}$.
      – Otherwise, Sim broadcasts $(\mathsf{eliminate}, P_i)$ on behalf of each honest party, marks P_i "$\mathsf{eliminated}$", and follows the protocol to update $\mathcal{P}_S$. All the messages sent from P_i during Π are set to 0 while emulating $\mathcal{A}'$ to interact with $\mathsf{Sim}_{\mathsf{GLOS}}$.
7. For each party P that is marked "$\mathsf{eliminated}$", all the messages sent from P_i during Π are set to 0 while emulating $\mathcal{A}'$ to interact with $\mathsf{Sim}_{\mathsf{GLOS}}$.

Figure 27: The simulator for the step of verifying the sharings.

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

For each virtual server V_j emulated by servers in $\mathcal{P}_S$, for each message with message index mid_m sent from a server $S_{j,\alpha}$ to another server $S_{j,\beta}$ during $\Pi_{\mathsf{Gb}}(V_j)$, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to receive $(\mathsf{send}_{S_{j,\alpha},S_{j,\beta}}, \mathsf{mid}_m)$ from the corrupted parties. Then:

– If $S_{j,\alpha}, S_{j,\beta}$ are both honest, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to send $(\mathsf{committed}, \mathsf{mid}_m)$ to each corrupted party.

– If $S_{j,\alpha}$ is honest and $S_{j,\beta}$ is corrupted, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to send $(\mathsf{committed}, \mathsf{mid}_m)$ to each corrupted party. In addition, Sim runs $\mathsf{Sim}_{\mathsf{GLOS}}$ to generate the message m and emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to send it to $S_{j,\beta}$.

– If $S_{j,\beta}$ is honest and $S_{j,\alpha}$ is corrupted, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to receive the message m from $S_{j,\alpha}$. Then, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to receive the request from $\mathcal{A}$.
   * If the request is $(\mathsf{deliver}, \mathsf{mid}_m)$, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to send $(\mathsf{committed}, \mathsf{mid}_m)$ to each corrupted party. Sim sends m to $\mathsf{Sim}_{\mathsf{GLOS}}$ on behalf of $\mathcal{A}'$ as the corresponding message in the execution of Π.
   * Otherwise, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to send $(\mathsf{eliminated}, S_{j,\alpha})$ to each corrupted party and follows the protocol to update $\mathcal{P}_S$. After this step, all the messages sent from $S_{j,\alpha}$ during the protocol Π are set to 0 while emulating $\mathcal{A}'$ to interact with $\mathsf{Sim}_{\mathsf{GLOS}}$.

– If $S_{j,\alpha}, S_{j,\beta}$ are both corrupted, $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to receive the message m from $S_{j,\alpha}$. Then, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to receive the request from $\mathcal{A}$.
   * If the request is $(\mathsf{deliver}, \mathsf{mid}_m)$, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to send $(\mathsf{committed}, \mathsf{mid}_m)$ to each corrupted party.
   * Otherwise, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to send $(\mathsf{eliminated}, S_{j,\alpha})$ to each corrupted party and follows the protocol to update $\mathcal{P}_S$. After this step, all the messages sent from $S_{j,\alpha}$ during Π are set to 0 while emulating $\mathcal{A}'$ to interact with $\mathsf{Sim}_{\mathsf{GLOS}}$.

Then, Sim checks whether each corrupted server $S_{j,\alpha}$ executes $\Pi_{\mathsf{Gb}}(V_j)$ honestly from their committed inputs (the shares are received by $\mathcal{F}_{\mathsf{VP2P}}$ in Step 3, and the ROT outputs are received by $\mathcal{F}_{\mathsf{BasedROT}}$ in Step 5 of Sim) and randomness (set in Step 7 of Sim). If not, Sim sets $\mathsf{Comp}_j = 1$.

Figure 28: The simulator for the step of garbling the computations.

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

1. Sim emulates $\mathcal{F}_{\mathsf{Coin}}$ to receive $\mathsf{RandomCoin}$ from the corrupted parties and faithfully emulates $\mathcal{F}_{\mathsf{Coin}}$ to sample r_{ver}. Then, Sim emulates $\mathcal{F}_{\mathsf{Coin}}$ to send it to all the parties. If r_{ver} has been queried by $\mathcal{A}$, Sim aborts the simulation. Otherwise, Sim sets the RO output $\mathcal{O}(r_{\mathsf{ver}})$ to $\mathcal{T}_{\mathsf{ver}}$.
2. For each $V_j \in \mathcal{T}_{\mathsf{ver}}$ and each message (with index mid_m) sent from $S_{j,\alpha}$ to $S_{j,\beta}$ during $\Pi_{\mathsf{Gb}}(V_j)$, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to receive $(\mathsf{open}_{S_{j,\alpha}}, \mathsf{mid}_m)$ (resp. $(\mathsf{open}_{S_{j,\beta}}, \mathsf{mid}_m)$) from the corrupted parties. Then:
   - If $S_{j,\alpha}$ and $S_{j,\beta}$ are both honest, Sim obtains m from $\mathsf{Sim}_{\mathsf{GLOS}}$ and emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to send m to all the corrupted parties.
   - If $S_{j,\alpha}$ (resp. $S_{j,\beta}$) is honest and $S_{j,\beta}$ (resp. $S_{j,\alpha}$) is corrupted, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to send the message m with index mid_m to all the corrupted parties.
   - If $S_{j,\alpha}$ (resp. $S_{j,\beta}$) is corrupted, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$ to receive a request from $\mathcal{A}$:
     - If $(\mathsf{open}, \mathsf{mid}_m, \mathsf{Set})$ is received, Sim broadcasts $(\mathsf{eliminate}, S_{j,\alpha})$ (resp. $(\mathsf{eliminate}, S_{j,\beta})$) on behalf of each honest party out of Set. If there are at least $t+1$ parties who broadcast $(\mathsf{eliminate}, S_{j,\alpha})$ (resp. $(\mathsf{eliminate}, S_{j,\beta})$, including the broadcasts received from corrupted parties and the broadcasts of honest parties emulated by Sim), Sim marks $S_{j,\alpha}$ (resp. $S_{j,\beta}$) "$\mathsf{eliminated}$" and follows the protocol to update $\mathcal{P}_S$,.
     - Otherwise, Sim broadcasts $(\mathsf{eliminate}, S_{j,\alpha})$ (resp. $(\mathsf{eliminate}, S_{j,\beta})$) on behalf of each honest party, marks $S_{j,\alpha}$ (resp. $S_{j,\beta}$) "$\mathsf{eliminated}$", and follows the protocol to update $\mathcal{P}_S$.
3. For each $V_j \in \mathcal{T}_{\mathsf{ver}}$ and each message (with index mid_m) sent from a server S to a server $S_{j,\alpha}$ emulating V_j during Π_{Share}, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S, S_{j,\alpha})$ to receive $(\mathsf{open}_{S_{j,\alpha}}, \mathsf{mid}_m)$ from the corrupted parties. Then:
   - If S and $S_{j,\alpha}$ are both honest, Sim obtains m from $\mathsf{Sim}_{\mathsf{GLOS}}$ and emulates $\mathcal{F}_{\mathsf{VP2P}}(S, S_{j,\alpha})$ to send m to all the corrupted parties.
   - If $S_{j,\alpha}$ is honest and S is corrupted, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S, S_{j,\alpha})$ to send the message m with index mid_m to all the corrupted parties.
   - If $S_{j,\alpha}$ is corrupted, Sim emulates $\mathcal{F}_{\mathsf{VP2P}}(S, S_{j,\alpha})$ to receive a request from $\mathcal{A}$:
     - If $(\mathsf{open}, \mathsf{mid}_m, \mathsf{Set})$ is received, Sim broadcasts $(\mathsf{eliminate}, S_{j,\alpha})$ on behalf of each honest party out of Set. If there are at least $t+1$ parties who broadcast $(\mathsf{eliminate}, S_{j,\alpha})$ (including the broadcasts received from corrupted parties and the broadcasts of honest parties emulated by Sim), Sim marks $S_{j,\alpha}$ "$\mathsf{eliminated}$" and follows the protocol to update $\mathcal{P}_S$.
     - Otherwise, Sim broadcasts $(\mathsf{eliminate}, S_{j,\alpha})$ on behalf of each honest party, marks $S_{j,\alpha}$ "$\mathsf{eliminated}$", and follows the protocol to update $\mathcal{P}_S$.
4. For each $V_j \in \mathcal{T}_{\mathsf{ver}}$, for each ROT correlations sent from $\mathcal{F}_{\mathsf{BaseROT}}$ to two members $(S_{j,\alpha}, S_{j,\beta})$ of V_j, Sim runs $\mathsf{Sim}_{\mathsf{GLOS}}$ to obtain (r_0, r_1) if $S_{j,\alpha}$ is honest and obtain (b, r_b) if $S_{j,\beta}$ is honest. Then, Sim samples the VSSs of the outputs from $\mathcal{F}_{\mathsf{BaseROT}}$ based on the secrets and corrupted servers' shares. Finally, Sim sends each honest server's shares to all the corrupted servers on behalf of the honest server, and Sim receives the corrupted servers' shares of these VSSs.
5. For each $V_j \in \mathcal{T}_{\mathsf{ver}}$:
   (a) For each honest server $S_{j,\alpha}$ emulating V_j, Sim runs $\mathsf{Sim}_{\mathsf{GLOS}}$ to obatin $S_{j,\alpha}$'s output $\mathsf{gbout}_{j,\alpha}$ from $\Pi_{\mathsf{Gb}}(V_j)$. Then, Sim randomly samples a κ-bit string as $r_{j,\alpha}$ and sends $(\mathsf{gbout}_{j,\alpha}, r_{j,\alpha})$ to all the servers. If $\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ has been queried by $\mathcal{A}$, Sim aborts the simulation. Otherwise, Sim sets the RO output for the query $\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ to the previously sampled commitment $\mathcal{O}(\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$ in Step 8.(b) of Sim.
   (b) For each corrupted server $S_{j,\alpha}$ emulating V_j, Sim receives $(\mathsf{gbout}_{j,\alpha}, r_{j,\alpha})$ from $S_{j,\alpha}$ and follows the protocol to check the commitments on behalf of each honest server. If the check fails, Sim follows the protocol to broadcast $(\mathsf{eliminate}, S_{j,\alpha})$ on behalf of the honest server. If the check passes, but the pair of $(\mathsf{eliminate}, S_{j,\alpha})$ received from $S_{j,\alpha}$ does not match the pair of $(\mathsf{gbout}_{j,\alpha}, r_{j,\alpha})$ set by Sim in Step 8.(c), Sim aborts the simulation.
6. For each $V_j \in \mathcal{T}_{\mathsf{ver}}$:
   (a) For each honest server $S_{j,\alpha}$ emulating V_j, Sim runs $\mathsf{Sim}_{\mathsf{GLOS}}$ to obtain $S_{j,\alpha}$'s randomness seed $\mathsf{seed}_{j,\alpha}$ for $\Pi_{\mathsf{Gb}}(V_j)$ and the random string $s_{j,\alpha}$ for generating the commitment to $\mathsf{seed}_{j,\alpha}$. If $\mathsf{seed}_{j,\alpha} \| s_{j,\alpha} \| j \| \alpha$ has

been queried by $\mathcal{A}$, Sim aborts the simulation. Otherwise, Sim sets the RO output for the query $\mathsf{seed}_{j,\alpha}\|s_{j,\alpha}\|j\|\alpha$ to the previously sampled commitment $\mathcal{O}(\mathsf{seed}_{j,\alpha}\|s_{j,\alpha}\|j\|\alpha)$ in Step 7 of Sim.

(b) For each corrupted server $S_{j,\alpha}$ emulating V_j, Sim receives $\mathsf{seed}_{j,\alpha}, s_{j,\alpha}$ from $S_{j,\alpha}$ and follows the protocol to check the commitments on behalf of each honest server. If the check fails, Sim follows the protocol to broadcast $(\mathsf{eliminate}, S_{j,\alpha})$ on behalf of the honest server. If the check passes, but the seed $\mathsf{seed}_{j,\alpha}$ received from $S_{j,\alpha}$ does not match the seed $\mathsf{seed}_{j,\alpha}$ set by Sim in Step 7, Sim aborts the simulation.

7. Sim follows the protocol to check each corrupted server $S_{j,\alpha}$'s view in $\{\Pi_{\mathsf{Gb}}(V_j)\}_{V_j\in\mathcal{T}_{\mathsf{ver}}}$ and Π_{ShVer} on behalf of each honest server. If the check fails, Sim broadcasts $(\mathsf{eliminate}, S_{j,\alpha})$ on behalf of the honest server.
8. When $\mathsf{Rerun}=0$, if for more than $N/160$ active virtual servers V_j, it holds that $\mathsf{Comp}_j=1$, Sim aborts the simulation. When $\mathsf{Rerun}=1$, if for more than $N'/240$ active virtual servers V_j, it holds that $\mathsf{Comp}_j=1$, Sim aborts the simulation.
9. For each active virtual server V_j with $\mathsf{Comp}_j=0$ and each corrupted party P that is not eliminated, Sim computes V_j's share of each sharing generated by P during Π_{Share} (by reconstructing the secret of the additive sharing of this share among $S_{j,1},\ldots,S_{j,c}$, where each message is received by $\mathcal{F}_{\mathsf{VP2P}}(P,S_{j,\alpha})$ for some $\alpha\in\{1,\ldots,c\}$). Then, Sim checks whether the shares are from valid sharings. If not, Sim aborts the simulation.
10. Let $\mathcal{T}_{\mathsf{cor}}$ be the set of corrupted virtual servers. Let $\mathcal{T}_{\mathsf{act}}$ be the set of active virtual servers. Let $\mathcal{T}_{\mathsf{hon}}$ be the set of virtual servers V_j with $\mathsf{Comp}_j=0$ in $\mathcal{T}_{\mathsf{act}}\backslash\mathcal{T}_{\mathsf{ver}}$. Then, Sim sends $\mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}$ to $\mathsf{Sim}_{\mathsf{GLOS}}$.

Figure 29: The simulator for the step of verifying the computations.

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

1. For each virtual server $V_j\in\mathcal{T}_{\mathsf{act}}$:
   (a) Sim obtains $\mathsf{gbout}_{j,\alpha}$ from $\mathsf{Sim}_{\mathsf{GLOS}}$ for each honest server $S_{j,\alpha}$ among $S_{j,1},\ldots,S_{j,c}$. Sim then samples a random κ-bit string as $r_{j,\alpha}$. If $\mathsf{gbout}_{j,\beta}\|r_{j,\beta}\|\mathsf{mid}_{\mathsf{gbout}_{j,\beta}}$ has been queried by $\mathcal{A}$, Sim aborts the simulation. Otherwise, Sim sends $\mathsf{gbout}_{j,\alpha}, r_{j,\alpha}$ to each corrupted server among $S_{j,1},\ldots,S_{j,c}$ on behalf of $S_{j,\alpha}$. Then, Sim sets the RO output for the query $\mathsf{gbout}_{j,\alpha}\|r_{j,\alpha}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ to the previously sampled commitment $\mathcal{O}(\mathsf{gbout}_{j,\alpha}\|r_{j,\alpha}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$ in Step 8.(b) of Sim.
   (b) Sim receives $\mathsf{gbout}_{j,\beta}, r_{j,\beta}$ from each corrupted server $S_{j,\beta}$ among $S_{j,1},\ldots,S_{j,c}$. Sim then follows the protocol to check whether $\mathcal{O}(\mathsf{gbout}_{j,\beta}\|r_{j,\beta}\|\mathsf{mid}_{\mathsf{gbout}_{j,\beta}})$ matches the commitment for $S_{j,\beta}$'s output. If not, Sim broadcasts $(\mathsf{complain}, \mathsf{mid}_{\mathsf{gbout}_{j,\beta}}, S_{j,\alpha}, S_{j,\beta})$ on behalf of each honest server among $S_{j,1},\ldots,S_{j,c}$. Otherwise, Sim sends $\mathsf{gbout}_{j,\beta}, r_{j,\beta}$ to all the clients on behalf of each honest server among $S_{j,1},\ldots,S_{j,c}$.
   (c) If the check passes, but $(\mathsf{gbout}_{j,\beta}, r_{j,\beta})$ received from $S_{j,\beta}$ does not match the pair of $(\mathsf{gbout}_{j,\beta}, r_{j,\beta})$ set in Step 8 of Sim, Sim aborts the simulation.
   (d) If either Sim or a corrupted server $S_{j,\alpha}$ broadcasts $(\mathsf{complain}, \mathsf{mid}, S_{j,\alpha}, S_{j,\beta})$:
   - If $S_{j,\beta}$ is honest (in this case, $S_{j,\alpha}$ must be corrupted):
     i. Sim sends $\mathsf{gbout}_{j,\beta}, r_{j,\beta}$ to all the servers on behalf of $S_{j,\beta}$.
     ii. Sim delivers $\mathsf{gbout}_{j,\beta}, r_{j,\beta}$ to $S_{j,\alpha}$ on behalf of each honest server.
   - If $S_{j,\beta}$ is corrupted:
     i. Sim receives $\mathsf{gbout}_{j,\beta}, r_{j,\beta}$ from $S_{j,\beta}$ on behalf of each honest server.
     ii. Sim then follows the protocol to check whether $\mathcal{O}(\mathsf{gbout}_{j,\beta}\|r_{j,\beta}\|\mathsf{mid}_{\mathsf{gbout}_{j,\beta}})$ matches the commitment for $S_{j,\beta}$'s output on behalf of each honest server. If not, Sim broadcasts $(\mathsf{eliminate}, S_{j,\beta})$ on behalf of the server.
     iii. If an honest server's check passes, but $(\mathsf{gbout}_{j,\beta}, r_{j,\beta})$ received by the server from $S_{j,\beta}$ does not match the pair of $(\mathsf{gbout}_{j,\beta}, r_{j,\beta})$ set in Step 8 of Sim, Sim aborts the simulation.
     iv. If $(\mathsf{eliminate}, S_{j,\beta})$ is received from $\geq t+1$ servers' broadcasts (including the broadcasts by Sim on behalf of honest parties and the broadcasts from corrupted servers), Sim follows the protocol to eliminate $S_{j,\beta}$ from $\mathcal{P}_S$. Otherwise, if $S_{j,\alpha}$ is corrupted, Sim delivers $\mathsf{gbout}_{j,\beta}, r_{j,\beta}$ received by each honest server whose check passes to $S_{j,\alpha}$ on behalf of the honest server.

Figure 30: The simulator for the step of collecting garbled circuits.

We construct the following hybrids:

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

$\mathbf{Hyb}_1$: In this hybrid, after emulating $\mathcal{F}_{\mathsf{VSS}}$ to receive corrupted clients' inputs, Sim sends them to $\mathcal{F}$ to obtain corrupted clients' outputs. This does not affect the output distribution. Thus, $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_0$ have the same distribution.

$\mathbf{Hyb}_2$: In this hybrid, Sim additionally starts the execution of Π in his head. The inputs are set to the servers' VSS shares of the input sharings, and the environment for Π is denoted by $\mathcal{Z}'$ (where the adversary for Π is denoted by $\mathcal{A}'$). Sim runs the honest parties in Π honestly. After r_{vp} is sampled in Π_{GLOS}, Sim sets the value r_{vp} in Π to be the same as that in Π_{GLOS}. The further behaviors of $\mathcal{Z}'$ are determined later. Furthermore, Sim initializes $\mathsf{Comp}_j = 0$ for each $j = 1, \ldots, N$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_1$ have the same distribution.

$\mathbf{Hyb}_3$: In this hybrid, for each message with message index mid_m sent from a corrupted server S_i to another server S_j during Π_{Share}, Sim follows the protocol to receive the message m on behalf of $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$. Then, for the protocol Π running in his head, Sim emulates $\mathcal{A}'$ to send $(\mathsf{send}, S_i, S_j, m)$ to $\mathcal{F}_{\mathsf{emulate}}$ and then faithfully emulates $\mathcal{F}_{\mathsf{emulate}}$ to generate its output. For each message m sent from an honest server S_i to a corrupted server S_j via $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$, during the protocol Π running in his head, Sim emulates $\mathcal{F}_{\mathsf{emulate}}$ to send (S_i, m) to S_j as the corresponding message. For messages sent between honest servers, Sim emulates $\mathcal{F}_{\mathsf{emulate}}$ faithfully in Π with corresponding messages in the execution of Π_{GLOS}. This does not affect the output distribution. Thus, $\mathbf{Hyb}_3$ and $\mathbf{Hyb}_2$ have the same distribution.

$\mathbf{Hyb}_4$: In this hybrid, for each honest party P, Sim does not follow the protocol to generate the auxiliary sharings $[\boldsymbol{r}_1^P]_{\times\kappa'}, [\boldsymbol{r}_2^P]^{(2)}_{\times\kappa'}, [\boldsymbol{r}_3^P]^{(3)}_{\times\kappa'}, [\boldsymbol{o}^P]^{(2)}_{\times\kappa'}$. Instead, Sim first generates the shares for the corrupted virtual servers, and then Sim randomly samples the other virtual servers' shares based on these shares and the secrets. Since for all these sharings, the number of corrupted virtual servers is smaller than $N/60$ during the first execution of Π_{GLOS} and is smaller than $N'/120$ with an overwhelming probability. In both executions, the number is smaller than the parameter T for the LSSS Σ. Thus, the set of these virtual servers' shares is independent of the secret, so we only change the order of generating different virtual servers' shares of the sharings without changing their distributions with an overwhelming probability. Thus, the distributions of $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_3$ are statistically close.

$\mathbf{Hyb}_5$: In this hybrid, for each honest party P, Sim does not follow the protocol to generate the additive sharings sent from P. Instead, Sim first generates the corrupted servers' shares, and then randomly samples the honest servers' shares based on the corrupted servers' shares and the secret. For those additive sharings completely shared among corrupted servers, we do not change anything. For other sharings, the set of corrupted servers' shares is independent of the secret, so we only change the order of generating the honest and corrupted servers' shares of the sharings without changing their distributions. Thus, $\mathbf{Hyb}_5$ and $\mathbf{Hyb}_4$ have the same output distribution.

$\mathbf{Hyb}_6$: In this hybrid, when each corrupted party P sends a message m of the auxiliary sharings to a server $S_{j,\alpha}$, Sim follows the protocol to emulate $\mathcal{F}_{\mathsf{VP2P}}(P, S_{j,\alpha})$ to receive the request from $\mathcal{A}$. If $(\mathsf{deliver}, \mathsf{mid}_m)$ is not received, Sim additionally eliminates P from $\mathcal{P}_S$ if P is a server. After this step, all the messages sent from S_i during Π are set to 0 while emulating $\mathcal{A}'$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_6$ and $\mathbf{Hyb}_5$ have the same output distribution.

$\mathbf{Hyb}_7$: In this hybrid, after emulating $\mathcal{F}_{\mathsf{BaseROT}}$ to receive corrupted servers' outputs of $\mathcal{F}_{\mathsf{BaseROT}}$ from $\mathcal{A}$, Sim emulates $\mathcal{A}'$ to send the same ROT instances to $\mathcal{F}_{\mathsf{emulate}}$ during the execution of Π in his head and then honestly emulates $\mathcal{F}_{\mathsf{emulate}}$ to generate honest servers' outputs of $\mathcal{F}_{\mathsf{GLOSbaseROT}}$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_7$ and $\mathbf{Hyb}_6$ have the same distribution.

$\mathbf{Hyb}_8$: In this hybrid, during the step of committing the randomness, for each honest server $S_{j,\alpha}$, Sim does not follow the protocol to commit his random seed $\mathsf{seed}_{j,\alpha}$. Instead, Sim samples a random string as $\mathcal{O}(\mathsf{seed}_{j,\alpha} \| s_{j,\alpha} \| j \| \alpha)$. Since $\mathcal{O}(\mathsf{seed}_{j,\alpha} \| s_{j,\alpha} \| j \| \alpha)$ is uniformly random unless the adversary $\mathcal{A}$ has queried $\mathcal{O}$ with $\mathsf{seed}_{j,\alpha} \| s_{j,\alpha} \| j \| \alpha$, this only changes the output distribution when $\mathcal{A}$ has made the query. As the query contains a κ-bit random string $s_{j,\alpha}$, each query only has at most $2^{-\kappa}$ probability to match $\mathsf{seed}_{j,\alpha} \| s_{j,\alpha} \| j \| \alpha$. Taking the union bound of all $\mathsf{poly}(\kappa)$ possible queries by $\mathcal{A}$, the probability that $\mathcal{A}$ has made the query is

still negligible. Thus, the distributions of $\mathbf{Hyb}_8$ and $\mathbf{Hyb}_7$ are computationally indistinguishable.

$\mathbf{Hyb}_9$: In this hybrid, during the step of committing the randomness, for each corrupted server $S_{j,\alpha}$, after receiving $S_{j,\alpha}$'s broadcast $\mathcal{O}(\mathsf{seed}_{j,\alpha}\|s_{j,\alpha}\|j\|\alpha)$, Sim additionally checks whether it matches an output of $\mathcal{O}$ with a query of the form $\mathsf{seed}\|s\|j\|\alpha$. If there are two such seed, Sim aborts the simulation. If there is a single pair of (seed, s) such that $\mathcal{O}(\mathsf{seed}_{j,\alpha}\|s_{j,\alpha}\|j\|\alpha)$ matches $\mathcal{O}(\mathsf{seed}\|s\|j\|\alpha)$, Sim sets $(\mathsf{seed}_{j,\alpha}, s_{j,\alpha}$ to be the pair of (seed, s). If no such a pair of (seed, s), Sim sets $(\mathsf{seed}_{j,\alpha}, s_{j,\alpha}) = 0$. In the protocol Π running in Sim's head, Sim sets the randomness $\sigma_{j,\alpha}$ for committing $S_{j,\alpha}$'s input set $\Pi_{\mathsf{Gb}}(V_j)$ to $s_{j,\alpha}$, i.e. A' sends $(\mathsf{query}, S_{j,\alpha}, I_{j,\alpha}\|s_{j,\alpha})$ to $\mathcal{F}_{\mathsf{emulate}}$ on behalf of $S_{j,\alpha}$ while generating his commitment. This only changes the output distribution when there exists $(\mathsf{seed}^{(1)}, s^{(1)}) \neq (\mathsf{seed}^{(2)}, s^{(2)}) \in (\{0,1\}^\kappa)^2$ such that $\mathcal{O}(\mathsf{seed}^{(1)}\|s^{(1)}\|j\|\alpha) = \mathcal{O}(\mathsf{seed}^{(2)}\|s^{(2)}\|j\|\alpha)$, where $\mathsf{seed}^{(1)}\|s^{(1)}\|j\|\alpha, \mathsf{seed}^{(2)}\|s^{(2)}\|j\|\alpha$ are both queried by $\mathcal{A}$. Note that each RO output is uniformly random to $\mathcal{A}$ before he queried it, so each pair of queries has only $2^{-\kappa}$ probability to lead to the same RO output. Taking the union bound of all $\mathsf{poly}(\kappa)$ pairs of queries by $\mathcal{A}$, the probability that the RO has the same output for two different queries made by $\mathcal{A}$ is still negligible. Thus, the distributions of $\mathbf{Hyb}_9$ and $\mathbf{Hyb}_8$ are computationally indistinguishable.

Note that $s_{j,\alpha}$ for each honest server $S_{j,\alpha}$ is not used until Π_{CompVer}, so Sim delays the generation of it to Π_{CompVer}.

$\mathbf{Hyb}_{10}$: In this hybrid, for each message with message index mid_m sent from a corrupted server $S_{j,\alpha}$ to another server $S_{j,\beta}$ during $\Pi_{\mathsf{Gb}}(V_j)$, Sim follows the protocol to receive the message m on behalf of $\mathcal{F}_{\mathsf{VP2P}}(S_{j,\alpha}, S_{j,\beta})$. Then, for the protocol Π running in his head, Sim emulates $\mathcal{A}'$ to send $(\mathsf{send}, S_i, S_j, m)$ to $\mathcal{F}_{\mathsf{emulate}}$ and then faithfully emulates $\mathcal{F}_{\mathsf{emulate}}$ to generate its output. For each message m sent from an honest server S_i to a corrupted server S_j via $\mathcal{F}_{\mathsf{VP2P}}(S_i, S_j)$, during the protocol Π running in his head, Sim emulates $\mathcal{F}_{\mathsf{emulate}}$ to send (S_i, m) to S_j as the corresponding message. For messages sent between honest servers, Sim emulates $\mathcal{F}_{\mathsf{emulate}}$ faithfully in Π with corresponding messages in the execution of Π_{GLOS}. This also does not affect the output distribution. Thus, $\mathbf{Hyb}_{10}$ and $\mathbf{Hyb}_9$ have the same distribution.

$\mathbf{Hyb}_{11}$: In this hybrid, after simulating the execution of $\Pi_{\mathsf{Gb}}(V_j)$, Sim checks whether each corrupted server $S_{j,\alpha}$ executes $\Pi_{\mathsf{Gb}}(V_j)$ honestly from their committed inputs (the shares are received by $\mathcal{F}_{\mathsf{VP2P}}$, and the ROT outputs are received by $\mathcal{F}_{\mathsf{BasedROT}}$) and randomness (set in $\mathbf{Hyb}_9$). If not, Sim sets $\mathsf{Comp}_j = 1$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{11}$ and $\mathbf{Hyb}_{10}$ have the same distribution.

$\mathbf{Hyb}_{12}$: In this hybrid, after simulating the execution of $\Pi_{\mathsf{Gb}}(V_j)$ (where V_j remains active), for each honest server $S_{j,\alpha}$ emulating V_j, Sim does not follow the protocol to commit his output $\mathsf{gbout}_{j,\alpha}$. Instead, Sim samples a random string as $\mathcal{O}(\mathsf{gbout}_{j,\alpha}\|r_{j,\alpha}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$. Since $\mathcal{O}(\mathsf{gbout}_{j,\alpha}\|r_{j,\alpha}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$ is uniformly random unless the adversary $\mathcal{A}$ has queried $\mathcal{O}$ with the query $\mathsf{gbout}_{j,\alpha}\|r_{j,\alpha}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$, this only changes the output distribution when $\mathcal{A}$ has made the query. As the query contains a κ-bit random string $r_{j,\alpha}$, each query only has at most $2^{-\kappa}$ probability to match $\mathsf{gbout}_{j,\alpha}\|r_{j,\alpha}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$. Taking the union bound of all $\mathsf{poly}(\kappa)$ possible queries by $\mathcal{A}$, the probability that $\mathcal{A}$ has made the query is still negligible. Thus, the distributions of $\mathbf{Hyb}_{12}$ and $\mathbf{Hyb}_{11}$ are computationally indistinguishable.

Note that $r_{j,\alpha}$ for each honest server $S_{j,\alpha}$ is not used until Π_{CompVer}, so Sim delays its generation until Π_{CompVer}.

$\mathbf{Hyb}_{13}$: In this hybrid, after simulating the execution of $\Pi_{\mathsf{Gb}}(V_j)$ (where V_j remains active), for each corrupted server $S_{j,\alpha}$, after receiving $S_{j,\alpha}$'s broadcast $\mathcal{O}(\mathsf{gbout}_{j,\alpha}\|r_{j,\alpha}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$, Sim additionally checks whether it matches an output of $\mathcal{O}$ with a query of the form $\mathsf{gbout}\|r\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$. If there are two such pairs of (gbout, r), Sim aborts the simulation. If there is a single pair of (gbout, r) such that $\mathcal{O}(\mathsf{gbout}_{j,\alpha}\|r_{j,\alpha}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$ matches $\mathcal{O}(\mathsf{gbout}\|r\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$, Sim sets $(\mathsf{gbout}_{j,\alpha}, r_{j,\alpha})$ to be (gbout, r). If there is no such pair of (gbout, r), $(\mathsf{gbout}_{j,\alpha}, r_{j,\alpha})$ is set to 0. This only changes the output distribution when there exist two pairs $(\mathsf{gbout}^{(1)}, r^{(1)}) \neq (\mathsf{gbout}^{(2)}, r^{(2)})$ such that

$$\mathcal{O}(\mathsf{gbout}^{(1)}\|r^{(1)}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}) = \mathcal{O}(\mathsf{gbout}^{(2)}\|r^{(2)}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}),$$

where $\mathsf{gbout}^{(1)}\|r^{(1)}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}, \mathsf{gbout}^{(2)}\|r^{(2)}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ are both queried by $\mathcal{A}$. Note that each RO output is uniformly random to $\mathcal{A}$ before he queried it, so each pair of queries has only $2^{-\kappa}$ probability of leading to the same RO output. Taking the union bound of all $\mathsf{poly}(\kappa)$ pairs of queries by $\mathcal{A}$, the probability that the

RO has the same output for two different queries made by $\mathcal{A}$ is still negligible. Thus, the distributions of $\mathbf{Hyb}_{13}$ and $\mathbf{Hyb}_{12}$ are computationally indistinguishable.

$\mathbf{Hyb}_{14}$: In this hybrid, while simulating Π_{ShVer}, after emulating $\mathcal{F}_{\mathsf{Coin}}$ to sample s, if s has been queried by $\mathcal{A}$, Sim aborts the simulation. Since s is a uniformly random element in $\mathbb{F}_{2^\kappa}$. Each previous query made by $\mathcal{A}$ has at most $2^{-\kappa}$ probability to match s. Taking the union bound of all $\mathsf{poly}(\kappa)$ pairs of queries by $\mathcal{A}$, the probability that $\mathcal{A}$ has queried $\mathcal{O}$ by s is still negligible. Thus, the distributions of $\mathbf{Hyb}_{14}$ and $\mathbf{Hyb}_{13}$ are computationally indistinguishable.

$\mathbf{Hyb}_{15}$: In this hybrid, while simulating Π_{ShVer}, for each honest party P, Sim does not sample each honest server $S_{j,\alpha}$'s shares of (the additive sharing of) the auxiliary sharings $[\boldsymbol{r}_1^P]_{\times\kappa'}, [\boldsymbol{r}_2^P]^{(2)}_{\times\kappa'}, [\boldsymbol{r}_3^P]^{(3)}_{\times\kappa'}, [\boldsymbol{o}^P]^{(2)}_{\times\kappa'}$ first and then follows the protocol to compute $S_{j,\alpha}$'s shares of

$$\langle([\tau_1^P]_{\kappa'})^{V_j}\rangle, \langle([\tau_2^P]^{(2)}_{\kappa'})^{V_j}\rangle, \langle([\tau_3^P]^{(3)}_{\kappa'})^{V_j}\rangle, \langle([\tau_4^P]^{(2)}_{\kappa'})^{V_j}\rangle.$$

Instead, Sim first samples the whole sharings $[\tau_1^P]_{\kappa'}, [\tau_2^P]^{(2)}_{\kappa'}, [\tau_3^P]^{(3)}_{\kappa'}, [\tau_4^P]^{(2)}_{\kappa'}$ based on the shares for corrupted virtual servers. Then, Sim samples the additive sharings $\langle([\tau_1^P]_{\kappa'})^{V_j}\rangle, \langle([\tau_2^P]^{(2)}_{\kappa'})^{V_j}\rangle, \langle([\tau_3^P]^{(3)}_{\kappa'})^{V_j}\rangle, \langle([\tau_4^P]^{(2)}_{\kappa'})^{V_j}\rangle$ based on the corrupted servers' shares and the secrets for each other V_j. Note that the shares of (the additive sharing of) the auxiliary sharings $[\boldsymbol{r}_1^P]_{\times\kappa'}, [\boldsymbol{r}_2^P]^{(2)}_{\times\kappa'}, [\boldsymbol{r}_3^P]^{(3)}_{\times\kappa'}, [\boldsymbol{o}^P]^{(2)}_{\times\kappa'}$ for honest servers are sampled in the same way, and they are added into the sharings $[\tau_1^P]_{\kappa'}, [\tau_2^P]^{(2)}_{\kappa'}, [\tau_3^P]^{(3)}_{\kappa'}, [\tau_4^P]^{(2)}_{\kappa'}$. We only change the order of generating the honest servers' shares of (the additive sharing of) the auxiliary sharings $[\boldsymbol{r}_1^P]_{\times\kappa'}, [\boldsymbol{r}_2^P]^{(2)}_{\times\kappa'}, [\boldsymbol{r}_3^P]^{(3)}_{\times\kappa'}, [\boldsymbol{o}^P]^{(2)}_{\times\kappa'}$ and the sharings $[\tau_1^P]_{\kappa'}, [\tau_2^P]^{(2)}_{\kappa'}, [\tau_3^P]^{(3)}_{\kappa'}, [\tau_4^P]^{(2)}_{\kappa'}$ without changing their distributions. Thus, $\mathbf{Hyb}_{15}$ and $\mathbf{Hyb}_{14}$ have the same distribution.

$\mathbf{Hyb}_{16}$: In this hybrid, while simulating Π_{ShVer}, for each party P, Sim follows the protocol to compute each corrupted server $S_{j,\alpha}$'s shares of the sharings $\langle([\tau_1^P]_{\kappa'})^{V_j}\rangle, \langle([\tau_2^P]^{(2)}_{\kappa'})^{V_j}\rangle, \langle([\tau_3^P]^{(3)}_{\kappa'})^{V_j}\rangle, \langle([\tau_4^P]^{(2)}_{\kappa'})^{V_j}\rangle$ from the committed messages by $\mathcal{F}_{\mathsf{VP2P}}(S, S_{j,\alpha})$ and checks whether they match the broadcast from $S_{j,\alpha}$. If not, Sim additionally sets $\mathsf{Comp}_j = 1$. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{16}$ and $\mathbf{Hyb}_{15}$ have the same distribution.

$\mathbf{Hyb}_{17}$: In this hybrid, while simulating Π_{ShVer}, after checking the sharings broadcast by each corrupted party P, if the sharings are not valid, Sim follows the protocol to update $\mathcal{P}_S$. Additionally, all the messages sent from P during the execution of Π in Sim's head are set to 0. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{17}$ and $\mathbf{Hyb}_{16}$ have the same distribution.

$\mathbf{Hyb}_{18}$: In this hybrid, while simulating Π_{ShVer}, if there are at least $t+1$ parties who broadcast $(\mathsf{eliminate}, S_i)$ (including the broadcasts received from corrupted parties and the broadcasts of honest parties emulated by Sim), Sim marks S_i "$\mathsf{eliminated}$" and follows the protocol to update $\mathcal{P}_S$. After this step, all the messages sent from S_i during the execution of Π in Sim's head are set to 0. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{18}$ and $\mathbf{Hyb}_{17}$ have the same distribution.

$\mathbf{Hyb}_{19}$: In this hybrid, while simulating Π_{CompVer}, if there are at least $t+1$ parties who broadcast $(\mathsf{eliminate}, S_i)$ (including the broadcasts received from corrupted parties and the broadcasts of honest parties emulated by Sim), Sim marks S_i "$\mathsf{eliminated}$" and follows the protocol to update $\mathcal{P}_S$. After this step, all the messages sent from S_i during the execution of Π in Sim's head are set to 0 after this step. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{19}$ and $\mathbf{Hyb}_{18}$ have the same distribution.

$\mathbf{Hyb}_{20}$: In this hybrid, we delay the generation of the honest servers' shares of the VSSs of the ROT correlations generated when Sim is emulating $\mathcal{F}_{\mathsf{BaseROT}}$ to the step of opening the ROT correlations during Π_{CompVer}. For those $V_j \notin \mathcal{T}_{\mathsf{ver}}$, the honest servers' shares of the VSSs of the ROT correlations are not used in the simulation, so Sim does not generate them. Note that these VSS shares are not used until the step of opening the ROT correlations during Π_{CompVer}, so delaying the generation of them does not affect the output distribution. $\mathbf{Hyb}_{20}$ and $\mathbf{Hyb}_{19}$ have the same distribution.

$\mathbf{Hyb}_{21}$: In this hybrid, while simulating Π_{CompVer}, for each $V_j \in \mathcal{T}_{\mathsf{ver}}$, after sampling $r_{j,\alpha}$ for each honest server $S_{j,\alpha}$ emulating V_j, Sim aborts the simulation if $\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ has been queried by $\mathcal{A}$. Otherwise, Sim sets the RO output for the query $\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ to the previously sampled commitment $\mathcal{O}(\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$.

If $\mathcal{A}$ has not queried $\mathcal{O}$ with $\mathsf{gbout}_{j,\alpha}\|r_{j,\alpha}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$, we do not change anything. On the other hand, $r_{j,\alpha}$ is uniformly randomly sampled, so each query from $\mathcal{A}$ can match $\mathsf{gbout}_{j,\alpha}\|r_{j,\alpha}\|\mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ with at most $2^{-\kappa}$ probability. Taking the union bound of all $\mathsf{poly}(\kappa)$ possible queries made by $\mathcal{A}$, the probability is still negligible. Thus, the distributions of $\mathbf{Hyb}_{21}$ and $\mathbf{Hyb}_{20}$ are computationally indistinguishable.

Note that for each virtual server $V_j \notin \mathcal{T}_{\mathsf{ver}}$, $r_{j,\alpha}$ is not used in the simulation until Π_{Collect} for each honest server $S_{j,\alpha}$ emulating V_j, so Sim delays the generation of it till Π_{Collect} in future hybrids.

$\mathbf{Hyb}_{22}$: In this hybrid, while simulating Π_{CompVer}, for each $V_j \in \mathcal{T}_{\mathsf{ver}}$, after sampling $s_{j,\alpha}$ for each honest server $S_{j,\alpha}$ emulating V_j, Sim aborts the simulation if $\mathsf{seed}_{j,\alpha}\|s_{j,\alpha}\|j\|\alpha$ has been queried by $\mathcal{A}$. Otherwise, Sim sets the RO output for the query $\mathsf{seed}_{j,\alpha}\|s_{j,\alpha}\|j\|\alpha$ to the previously sampled commitment $\mathcal{O}(\mathsf{seed}_{j,\alpha}\|s_{j,\alpha}\|j\|\alpha)$.

If $\mathcal{A}$ has not queried $\mathcal{O}$ with $\mathsf{seed}_{j,\alpha}\|s_{j,\alpha}\|j\|\alpha$, we do not change anything. On the other hand, $s_{j,\alpha}$ is uniformly randomly sampled, so each query from $\mathcal{A}$ can match $\mathsf{seed}_{j,\alpha}\|s_{j,\alpha}\|j\|\alpha$ with at most $2^{-\kappa}$ probability. Taking the union bound of all $\mathsf{poly}(\kappa)$ possible queries made by $\mathcal{A}$, the probability is still negligible. Thus, the distributions of $\mathbf{Hyb}_{22}$ and $\mathbf{Hyb}_{21}$ are computationally indistinguishable.

Note that for each virtual server $V_j \notin \mathcal{T}_{\mathsf{ver}}$, $s_{j,\alpha}$ is not used in the simulation for each honest server $S_{j,\alpha}$ emulating V_j, so Sim does not generate in future hybrids.

$\mathbf{Hyb}_{23}$: In this hybrid, after checking the opened views during Π_{CompVer}, when $\mathsf{Rerun} = 0$, if for more than $N/160$ active virtual servers V_j, it holds that $\mathsf{Comp}_j = 1$, Sim aborts the simulation. When $\mathsf{Rerun} = 1$, if for more than $N'/240$ active virtual servers V_j, it holds that $\mathsf{Comp}_j = 1$, Sim aborts the simulation.

Recall that we have proved in Section C that for any set of 4κ virtual servers, it has only $2^{-\Omega(\kappa)}$ probability to not be included in $\mathcal{T}_{\mathsf{ver}}$. Since for each $\mathsf{Comp}_j = 1$, there exists a corrupted server S that maliciously behaves while emulating a server $S_{j,\alpha}$, which will cause S to be eliminated from $\mathcal{P}$ if V_j is in $\mathcal{T}_{\mathsf{ver}}$. Thus, when $\mathsf{Rerun} = 0$, if for more than $N/160$ active virtual servers V_j, it holds that $\mathsf{Comp}_j = 1$, there must be a corrupted server that maliciously behaves while emulating the servers in more than $N/160t = 3\kappa$ virtual parties. The distribution only changes when the server S remains in $\mathcal{P}_S$ after the check. However, as we proved in Section C, the server S should be eliminated from $\mathcal{P}_S$ with an overwhelming probability, so the probability that S remains in $\mathcal{P}_S$ after the check is negligible. Similarly, when $\mathsf{Rerun} = 1$, if for more than $N'/240$ active virtual servers V_j, it holds that $\mathsf{Comp}_j = 1$, there must be a corrupted server that maliciously behaves while emulating the servers in more than $2N'/240t = 3\kappa$ virtual parties. In this case, the probability that S remains in $\mathcal{P}_S$ after the check is also negligible. Thus, the distributions of $\mathbf{Hyb}_{23}$ and $\mathbf{Hyb}_{22}$ are computationally indistinguishable.

$\mathbf{Hyb}_{24}$: In this hybrid, after checking the opened views during Π_{CompVer}, let $\mathcal{T}_{\mathsf{hon}}$ be the set of active honest virtual servers V_j with $\mathsf{Comp}_j = 0$ that are not in $\mathcal{T}_{\mathsf{ver}}$. For each active honest virtual server V_j with $\mathsf{Comp}_j = 0$, Sim computes V_j's share of each sharing (excluding the auxiliary sharings) generated by P (by reconstructing the secret of the additive sharing of this share among $S_{j,1}, \ldots, S_{j,c}$, where each message is received by $\mathcal{F}_{\mathsf{VP2P}}(P, S_{j,\alpha})$ for some $\alpha \in \{1, \ldots, c\}$). Then, Sim checks whether the shares are from valid sharings. If not, Sim aborts the simulation.

This only changes the output distribution when the shares of a sharing generated by a corrupted party P (that remains not eliminated) during Π_{Share} for the active virtual servers V_j with $\mathsf{Comp}_j = 0$ are not from a valid sharing.

Recall that for the servers emulating an active virtual server V_j with $\mathsf{Comp}_j = 0$, the linear combinations of the shares are honestly computed during Π_{ShVer}. Assume that the shares for virtual servers in $\mathcal{T}_3$ of a Σ-sharing generated by P during Π_{Share} are not valid. Assume that the random coefficient on this sharing in $[\tau_1^P]_{\kappa'} = \sum_{j=1}^{k_1} s_{1,j} \cdot [\boldsymbol{x}_j]_{\kappa'} + [\boldsymbol{r}_1^P]_{\kappa'}$ is $\sigma \in \mathbb{F}_{2^{\kappa'}}$. If $s_{1,1}, \ldots, s_{1,k_1} \in \mathbb{F}_{2^{\kappa'}}$ are all truly random, we can sample σ after the invalid sharing is fixed. If there exists $\sigma_0 \neq \sigma_0' \in \mathbb{F}_{2^{\kappa'}}$ such that $\sigma = \sigma_0$ and $\sigma = \sigma_0'$ both lead to a valid $[\tau_1^P]_{\kappa'}$, then the invalid sharing (which has been embedded in a $\Sigma_{\times\kappa'}$-sharing) is $(\sigma_0 - \sigma_0')^{-1}$ 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 $\sigma_0 \in \mathbb{F}_{2^{\kappa'}}$ that can make $[\tau]_{\kappa'}$ pass the check. The probability is $2^{-\kappa'}$. Considering the union bound for no more than 2^N possible choices of the set of active virtual servers with $\mathsf{Comp}_j = 0$, the probability is still no more than $2^{N-\kappa'} = 2^{-\kappa}$ if $s_{1,1}, \ldots, s_{1,k_1} \in \mathbb{F}_{2^{\kappa'}}$ are all truly random, which is negligible. Thus, if there is a non-negligible probability that $[\tau_1^P]_{\kappa'}$ is valid, then the truly random field elements and the

pseudo-random values $s_{1,1}, \ldots, s_{1,k_1} \in \mathbb{F}_{2^{\kappa'}}$ can be distinguished by computing $[\tau_1^P]_{\kappa'}$ with a non-negligible probability, which contradicts the definition of a PRG, so the probability that $[\tau_1^P]_{\kappa'}$ is valid is negligible.

Similarly, when the shares for virtual servers in $\mathcal{T}_3$ of a $\Sigma^{(2)}, \Sigma^{(3)}$-sharing or a $\Sigma^{(2)}$-sharing with an all-zero secret is not valid, the verification of the sharings only passes with a negligible probability. Therefore, the distribution only changes with a negligible probability. Thus, the distributions of $\mathbf{Hyb}_{24}$ and $\mathbf{Hyb}_{23}$ are computationally indistinguishable.

$\mathbf{Hyb}_{25}$: In this hybrid, after simulating Π_{Ver}, Sim sets $\mathcal{T}_{\mathsf{act}}$ to be the set of active virtual servers. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{25}$ and $\mathbf{Hyb}_{24}$ have the same distribution.

$\mathbf{Hyb}_{26}$: In this hybrid, at the beginning of the simulation of Π_{Collect}, for each virtual server $V_j \in \mathcal{T}_{\mathsf{act}}$, after sampling $r_{j,\alpha}$ for each honest server $S_{j,\alpha}$ emulating V_j, Sim aborts the simulation if $\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ has been queried by $\mathcal{A}$. Otherwise, Sim sets the RO output for the query $\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ to the previously sampled commitment $\mathcal{O}(\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}})$.

If $\mathcal{A}$ has not queried $\mathcal{O}$ with $\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$, we do not change anything. On the other hand, $r_{j,\alpha}$ is uniformly randomly sampled, so each query from $\mathcal{A}$ can match $\mathsf{gbout}_{j,\alpha} \| r_{j,\alpha} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ with at most $2^{-\kappa}$ probability. Taking the union bound of all $\mathsf{poly}(\kappa)$ possible queries made by $\mathcal{A}$, the probability is still negligible. Thus, the distributions of $\mathbf{Hyb}_{26}$ and $\mathbf{Hyb}_{25}$ are computationally indistinguishable.

Note that for each virtual server $V_j \notin \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}$, $r_{j,\alpha}$ is not used in the simulation for each honest server $S_{j,\alpha}$ emulating V_j, so Sim does not generate in future hybrids.

$\mathbf{Hyb}_{27}$: In this hybrid, while simulating Π_{Collect}, after checking $(\mathsf{gbout}_{j,\beta}, r_{j,\beta})$ on behalf of an honest server, Sim aborts the simulation if the pair of $(\mathsf{gbout}_{j,\beta}, r_{j,\beta})$ received from $S_{j,\beta}$ does not match the pair of $(\mathsf{gbout}_{j,\beta}, r_{j,\beta})$ set by Sim after $\mathsf{gbout}_{j,\beta}$ is committed but the check passes. Here we can simply assume that $\mathcal{A}$ made an additional random oracle query call of $(\mathsf{gbout}_{j,\beta}, r_{j,\beta})$ received from $\mathcal{A}$. Then, there exists $(\mathsf{gbout}^{(1)}, r^{(1)}) \neq (\mathsf{gbout}^{(2)}, r^{(2)})$ such that

$$\mathcal{O}(\mathsf{gbout}^{(1)} \| r^{(1)} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}) = \mathcal{O}(\mathsf{gbout}^{(2)} \| r^{(2)} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}),$$

where $\mathsf{gbout}^{(1)} \| r^{(1)} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}, \mathsf{gbout}^{(2)} \| r^{(2)} \| \mathsf{mid}_{\mathsf{gbout}_{j,\alpha}}$ are both queried by $\mathcal{A}$.

Note that each RO output is uniformly random to $\mathcal{A}$ before he queried it, so each pair of queries has only $2^{-\kappa}$ probability to lead to the same RO output. Taking the union bound of all $\mathsf{poly}(\kappa)$ pairs of queries by $\mathcal{A}$, the probability that the RO has the same output for two different queries made by $\mathcal{A}$ is still negligible. Thus, the distributions of $\mathbf{Hyb}_{27}$ and $\mathbf{Hyb}_{26}$ are computationally indistinguishable.

$\mathbf{Hyb}_{28}$: In this hybrid, after determining the virtual servers in each execution of Π, Sim sets $\mathcal{T}_{\mathsf{cor}}$ to be the set of corrupted virtual servers. If $|\mathcal{T}_{\mathsf{cor}}| > N/60$ and $\mathsf{Rerun} = 0$, Sim aborts the simulation. If $|\mathcal{T}_{\mathsf{cor}}| > N'/120$ and $\mathsf{Rerun} = 1$, Sim also aborts the simulation. This only changes the output distribution when $|\mathcal{T}_{\mathsf{cor}}| > N/60, \mathsf{Rerun} = 0$ or $|\mathcal{T}_{\mathsf{cor}}| > N'/120$ and $\mathsf{Rerun} = 1$. From our analysis in Section C, this only happens with a negligible probability. Thus, the distributions of $\mathbf{Hyb}_{27}$ and $\mathbf{Hyb}_{26}$ are statistically close.

$\mathbf{Hyb}_{29}$: In this hybrid, Sim does not follow the protocol to determine the set $\mathcal{T}_{\mathsf{ver}}$. Instead, $\mathcal{T}_{\mathsf{ver}}$ is determined to be a random set of virtual servers right after r_{vp} is sampled. During the protocol Π running in Sim's head, the set $\mathcal{T}_{\mathsf{ver}}$ sent by $\mathcal{Z}'$ to the honest servers are set to the sampled set $\mathcal{T}_{\mathsf{ver}}$ by Sim. During the verification of virtual servers' computations, after r_{ver} is sampled, if it has been queried by the adversary, Sim aborts the simulation. Otherwise, Sim sets the RO output $\mathcal{O}(r_{\mathsf{ver}})$ to the previously sampled set $\mathcal{T}_{\mathsf{ver}}$. Under the condition that r_{ver} has not been queried before it is sent to the adversary, $\mathcal{T}_{\mathsf{ver}}$ is uniformly random in both hybrids. Therefore, the output distribution only changes when r_{ver} has been queried by $\mathcal{A}$ before it is sent to $\mathcal{A}$. Note that r_{ver} is a uniformly random κ-bit string, the probability that the $\mathsf{poly}(\kappa)$ queries made by $\mathcal{A}$ include r_{ver} is no more than $\mathsf{poly}(\kappa)/2^{\kappa}$, which is negligible. Thus, the distributions of $\mathbf{Hyb}_{29}$ and $\mathbf{Hyb}_{28}$ are computationally indistinguishable.

$\mathbf{Hyb}_{30}$: In this hybrid, after Π_{CompVer} completes, during the protocol Π running Sim's head, Sim lets $\mathcal{Z}'$ send $\mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}$ to all the servers after the garbling protocols in Π completes. Then, Sim emulates $\mathcal{F}_{\mathsf{emulate}}$ to send $\mathsf{success}$ to all the servers and follows the protocol to compute honest servers' outputs in Π. This does not affect the output distribution. Thus, $\mathbf{Hyb}_{30}$ and $\mathbf{Hyb}_{29}$ have the same distribution.

$\mathbf{Hyb}_{31}$: In this hybrid, Sim does not follow the protocol to compute the following:

- The random coin r_{vp} that determines the committees for virtual servers.
- The secrets of the sharings output by $\mathcal{F}_{\mathsf{BaseROT}}$ (i.e. the outputs from $\mathcal{F}_{\mathsf{GLOSbaseROT}}$ in Π).
- The parties' views during Π_{Share}.
- The seed $\mathsf{seed}_{j,\alpha}$ (together with the κ-bit string $\sigma_{j,\alpha}$ for committing to the seed) for generating the randomness of each server $S_{j,\alpha}$ used in $\Pi_{\mathsf{Gb}}(V_j)$.
- The servers' views (including their outputs) during $\Pi_{\mathsf{Gb}}(V_j)$ for each virtual server V_j.

Instead, Sim uses the corresponding messages computed during the execution of Π in his head as these messages.

Note that the computations of these honest parties' messages are the same as the execution of Π in Sim's head, so we do not change anything when generating these messages. In addition, we do not change the evaluation process of each honest party P's output. Therefore, making the above changes does not affect the output distribution. Thus, $\mathbf{Hyb}_{31}$ and $\mathbf{Hyb}_{30}$ have the same distribution.

$\mathbf{Hyb}_{32}$: In this hybrid, honest clients do not follow the execution of Π_{GLOS} to obtain their outputs. Instead, the honest clients get their outputs from a functionality $\mathcal{F}$ for the circuit C' to compute the outputs on the servers' VSS shares, where the corrupted servers' VSS shares are set to 0. For each honest client P, since we ensure that $\mathcal{E}_P$ has the property of 3-multiplicative e-error-correction for both executions of Π_{GLOS} (see our analysis in Section 6.6), P's output obtained from Π_{GLOS} can be computed correctly from Π_{Ev}, i.e., each honest client's output from Π_{GLOS} is the same as that from the functionality for C' with an overwhelming probability for some specific corrupted servers' inputs $\boldsymbol{x}'_{\mathcal{C}}$ to C'.

Note that the corrupted servers' inputs to the functionality do not affect P's output, as C' first reconstructs the secrets of the VSSs, which are fixed by the shares for honest servers, and then evaluates the circuit C. Therefore, no matter what $\boldsymbol{x}'_{\mathcal{C}}$ is, the output of P is the same, so setting the corrupted servers' VSS shares to 0 won't change the clients' outputs. Thus, $\mathbf{Hyb}_{32}$ and $\mathbf{Hyb}_{31}$ have the same distribution.

$\mathbf{Hyb}_{33}$: In this hybrid, honest clients do get their outputs from a functionality $\mathcal{F}$ for the circuit C' to compute the outputs on the servers' VSS shares. Instead, they get their outputs directly from $\mathcal{F}$ for the circuit C on their real inputs, where corrupted clients' inputs to C are reconstructed from the honest servers' shares of the input VSSs generated by corrupted clients. The computational process for honest clients' outputs in both hybrids is actually the same. Thus, $\mathbf{Hyb}_{33}$ and $\mathbf{Hyb}_{32}$ have the same distribution.

$\mathbf{Hyb}_{34}$: In this hybrid, Sim does not follow the protocol Π to compute the following:

- The random coin r_{vp} that determines the committees for virtual servers.
- Corrupted servers' outputs from $\mathcal{F}_{\mathsf{GLOSbaseROT}}$.
- Corrupted parties' views during Π_{Share}.
- The seed $\mathsf{seed}_{j,\alpha}$ (together with the κ-bit string $\sigma_{j,\alpha}$ for committing to the seed) for generating the randomness of each corrupted server $S_{j,\alpha}$ used in $\Pi_{\mathsf{Gb}}(V_j)$.
- Corrupted servers' views during $\Pi_{\mathsf{Gb}}(V_j)$ for each virtual server V_j.
- Honest servers' views during $\Pi_{\mathsf{Gb}}(V_j)$ for each virtual server $V_j \in \mathcal{T}_{\mathsf{ver}}$.
- Each honest server $S_{j,\alpha}$'s output $\mathsf{gbout}_{j,\alpha}$ from $\Pi_{\mathsf{Gb}}(V_j)$ for each V_j in the set $\mathcal{T}_{\mathsf{act}}$ determined after Π_{CompVer}.

Instead, Sim computes the above information by emulating $\mathcal{A}'$ to interact with $\mathsf{Sim}_{\mathsf{GLOS}}$, where the corrupted clients' outputs from $\mathcal{F}_{\mathsf{GLOS}}$ are set to the corrupted clients' outputs received from $\mathcal{F}$. In particular, Sim sets the set $\mathcal{T}_{\mathsf{ver}}$ to what it generates after sampling r_{vp}. Before generating each honest server $S_{j,\alpha}$'s output $\mathsf{gbout}_{j,\alpha}$ from $\Pi_{\mathsf{Gb}}(V_j)$ for each $V_j \in \mathcal{T}_{\mathsf{act}}$, Sim sends $\mathcal{T}_{\mathsf{act}}, \mathcal{T}_{\mathsf{hon}}$ to $\mathsf{Sim}_{\mathsf{GLOS}}$.

In addition, Sim no longer generates the following:

- The messages sent from an honest party to each honest server $S_{j,\alpha}$ emulating $V_j \notin \mathcal{T}_{\mathsf{ver}}$.
- The output of $\mathcal{T}_{\mathsf{GLOSbaseROT}}$ to each honest server $S_{j,\alpha}$ emulating $V_j \notin \mathcal{T}_{\mathsf{ver}}$.
- The seed $\mathsf{seed}_{j,\alpha}$ for generating the randomness of each honest server $S_{j,\alpha}$ used in $\Pi_{\mathsf{Gb}}(V_j)$ for each $V_j \notin \mathcal{T}_{\mathsf{ver}}$.
- The messages sent between two honest servers during $\Pi_{\mathsf{Gb}}(V_j)$ for each virtual server $V_j \notin \mathcal{T}_{\mathsf{ver}}$.
- Each honest server $S_{j,\alpha}$'s output $\mathsf{gbout}_{j,\alpha}$ from $\Pi_{\mathsf{Gb}}(V_j)$ for each $V_j \notin \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}$.

These messages are not used in the simulation now, so not generating them will not affect the output distribution.

Recall that we have guaranteed that for no more than $N/160$ active virtual servers V_j in the first execution of Π_{GLOS} (or the simulation should be aborted), it holds that $\mathsf{Comp}_j = 1$. That is to say, $|\mathcal{T}_{\mathsf{act}} \backslash (\mathcal{T}_{\mathsf{hon}} \cup \mathcal{T}_{\mathsf{ver}})| \leq N/160$. Moreover, we guarantee that $|\mathcal{T}_{\mathsf{cor}}| \leq N/60$. Therefore, it holds that $|\mathcal{T}_{\mathsf{cor}} \cup \mathcal{T}_{\mathsf{ver}} \cup \mathcal{T}_{\mathsf{act}}| - |\mathcal{T}_{\mathsf{hon}}| \leq N/160 + N/240 + N/60 < T$. Furthermore, we have ensured that the shares generated in Π_{Share} for virtual servers in $\mathcal{T}_{\mathsf{hon}}$ are from valid sharings, and the corrupted servers run $\Pi_{\mathsf{Gb}}(V_j)$ for each $V_j \in \mathcal{T}_{\mathsf{hon}}$ (which is guaranteed by $\mathsf{Comp}_j = 0$). Thus, the set $\mathcal{T}_{\mathsf{hon}}$ satisfies all the required properties, so $\mathcal{F}_{\mathsf{emulate}}$ must output $\mathsf{success}$. In this case, $\mathsf{Sim}_{\mathsf{GLOS}}$ is able to generate $S_{j,\alpha}$'s output $\mathsf{gbout}_{j,\alpha}$ from $\Pi_{\mathsf{Gb}}(V_j)$ for each $V_j \in \mathcal{T}_{\mathsf{act}}$. Note that the messages generated from $\mathsf{Sim}_{\mathsf{GLOS}}$ and the execution of Π are computationally indistinguishable. We then argue that $\mathbf{Hyb}_{34}$ and $\mathbf{Hyb}_{33}$ are computationally indistinguishable.

For the sake of contradiction, assume that $\mathbf{Hyb}_{34}$ and $\mathbf{Hyb}_{33}$ are computationally distinguishable. By the standard hybrid argument, there exists a PPT algorithm $\mathcal{D}$ that can distinguish the output distributions before and after we change the listed information computed from the execution of Π to those simulated by running $\mathsf{Sim}_{\mathsf{GLOS}}$ with a non-negligible probability. Now we construct a PPT algorithm $\mathcal{D}'$ that breaks the security of Π. Concretely, $\mathcal{D}'$ receives a string from the challenger, which is either the listed information computed from executing Π in Sim's head or running $\mathsf{Sim}_{\mathsf{GLOS}}$. Then $\mathcal{D}'$ runs Sim with the change that the listed messages are replaced by the information received from the challenger. Finally, $\mathcal{D}'$ uses $\mathcal{D}$ to distinguish whether the string received from the challenger is generated from Π or $\mathsf{Sim}_{\mathsf{GLOS}}$. Note that the advantage of $\mathcal{D}'$ is the same as that of $\mathcal{D}$, which breaks the security of Π. This leads to a contradiction. Thus, the distributions of $\mathbf{Hyb}_{34}$ and $\mathbf{Hyb}_{33}$ are computationally indistinguishable.

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

K The Cost Analysis for the Main Protocol

We analyze the communication cost of our main protocol Π_{Main} in this section.

1. **VSS Inputs.** This step only contains invocations of $\mathcal{F}_{\mathsf{VSS}}$. From Lemma 2, generating the VSS for each bit of the inputs to C requires communication of $\mathsf{poly}(n, \kappa)$ bits. Thus, this step requires communication of $W_I \cdot \mathsf{poly}(n, \kappa)$ bits (when $\mathcal{F}_{\mathsf{VSS}}$ is replaced by the protocol realizing it in the plain/RO model, similar below) via both P2P and broadcast channels.

2. **Determining Virtual Servers.** This step only contains an invocation of $\mathcal{F}_{\mathsf{Coin}}$. From Lemma 3, this only requires communication of $\mathsf{poly}(n, \kappa)$ bits via both P2P and broadcast channels.

3. **Generating Sharings.** In this step, the parties run Π_{Share} with each message sent via $\mathcal{F}_{\mathsf{VP2P}}$. For each message of size m sent during Π_{Share}, the communication cost of sending it is $O(|m|n + \mathsf{poly}(n, \kappa))$ bits via P2P channels and $\mathsf{poly}(n, \kappa)$ bits via broadcast channels. There are $\mathsf{poly}(n, \kappa)$ messages sent during Π_{Share}, so the communication cost of this step is $O(n \cdot \mathtt{CC}(\Pi_{\mathsf{Share}}) + \mathsf{poly}(n, \kappa))$ bits via P2P channels and $\mathsf{poly}(n, \kappa)$ bits via broadcast channels, where $\mathtt{CC}(\Pi_{\mathsf{Share}})$ is the communication cost of Π_{Share}.

4. **Generating Auxiliary Sharings.** In this step, each party P generates four sharings, where each sharing's size is $O(n\kappa') = \mathsf{poly}(n, \kappa)$ bits. Therefore, the total communication cost of this step is $O((\tilde{n} + n) \cdot \mathsf{poly}(n, \kappa))$.

5. **Preparing Base ROTs.** This step only contains an invocation of $\mathcal{F}_{\mathsf{BaseROT}}$. From Lemma 5, this step requires communication of $\mathsf{poly}(n,\kappa)$ bits via both P2P and broadcast channels.

6. **Verifying the Sharings.** In this step, the parties first invoke $\mathcal{F}_{\mathsf{Coin}}$, which requires communication of $\mathsf{poly}(n,\kappa)$ bits via both P2P and broadcast channels. Then, for each party P, the parties need to broadcast 2 Σ-sharings, 4 $\Sigma^{(2)}$-sharings, and 2 $\Sigma^{(3)}$-sharings, which requires communication of $\mathsf{poly}(n,\kappa)$ bits via broadcast channels. Finally, the parties' shares may need to be opened by $\mathcal{F}_{\mathsf{VP2P}}$, which requires communication of $O(n\cdot\mathtt{CC}(\Pi_{\mathsf{Share}})+\mathsf{poly}(n,\kappa))$ bits via P2P channels and $\mathsf{poly}(n,\kappa)$ bits via broadcast channels. The broadcasts of requests also require communication of $\mathsf{poly}(n,\kappa)$ bits via broadcast channels. In total, this step requires communication of $O(n\cdot\mathtt{CC}(\Pi_{\mathsf{Share}})+(n+\tilde{n})\cdot\mathsf{poly}(n,\kappa))$ bits via P2P channels and $\mathsf{poly}(n,\kappa)$ bits via broadcast channels.

7. **Committing to the Randomness.** In this step, each server $S_{j,\alpha}$ will broadcast a commitment of $\mathsf{poly}(n,\kappa)$ size. Thus, this step requires communication of $Nc\cdot\mathsf{poly}(n,\kappa)=\mathsf{poly}(n,\kappa)$ bits via broadcast channels.

8. **Garbling the Computations.** In this step, the servers run $\Pi_{\mathsf{Gb}}(V_j)$ for each virtual server V_j with each message sent via $\mathcal{F}_{\mathsf{VP2P}}$, which requires communication of $O(n\cdot\mathtt{CC}(\Pi_{\mathsf{Gb}})+\mathsf{poly}(n,\kappa))$ bits via P2P channels and $\mathsf{poly}(n,\kappa)$ bits via broadcast channels, where $\mathtt{CC}(\Pi_{\mathsf{Gb}})$ is the communication cost of $\Pi_{\mathsf{Gb}}(V_j)$ for all virtual servers V_j. In addition, each server $S_{j,\alpha}$ will broadcast a commitment of $\mathsf{poly}(n,\kappa)$ size, which requires communication of $Nc\cdot\mathsf{poly}(n,\kappa)=\mathsf{poly}(n,\kappa)$ bits via broadcast channels. In total, this step requires communication of $O(n\cdot\mathtt{CC}(\Pi_{\mathsf{Gb}})+\mathsf{poly}(n,\kappa))$ bits via P2P channels and $\mathsf{poly}(n,\kappa)$ bits via broadcast channels.

9. **Verifying the Computations.** In this step, the parties first invoke $\mathcal{F}_{\mathsf{Coin}}$, which requires communication of $\mathsf{poly}(n,\kappa)$ bits via both P2P and broadcast channels. Then, a small fraction of messages in the sharing and garbling phases of Π_{GLOS} needs to be opened, which requires $O(n\cdot(\mathtt{CC}(\Pi_{\mathsf{Share}})+\mathtt{CC}(\Pi_{\mathsf{Gb}}))+\mathsf{poly}(n,\kappa))$ bits via P2P channels and $\mathsf{poly}(n,\kappa)$ bits via broadcast channels. A fraction of the VSSs for base ROTs and the random seeds should also be opened, which requires communication of $\mathsf{poly}(n,\kappa)$ bits via P2P channels. The broadcasts of requests also require communication of $\mathsf{poly}(n,\kappa)$ bits via broadcast channels. In total, this step requires communication of $O(n\cdot(\mathtt{CC}(\Pi_{\mathsf{Share}})+\mathtt{CC}(\Pi_{\mathsf{Gb}}))+\mathsf{poly}(n,\kappa))$ bits via P2P channels and $\mathsf{poly}(n,\kappa)$ bits via broadcast channels.

10. **Collecting Garbled Circuits.** In this step, the outputs of $\Pi_{\mathsf{Gb}}(V_j)$ for $V_j\in\mathcal{T}_{\mathsf{act}}$ are opened to a constant number of servers. This step requires communication of $O(\mathtt{GCSIZE}+\mathsf{poly}(n,\kappa))$ bits via P2P channels, where $\mathtt{GCSIZE}$ is the total output size of $\Pi_{\mathsf{Gb}}(V_j)$ for all virtual parties V_j. Then, because of the complaints, the outputs may need to be sent to all the servers, so additional communication of $O(n\cdot\mathtt{GCSIZE}+\mathsf{poly}(n,\kappa))$ bits via P2P channels is required. Finally, sending the outputs to the clients requires communication of $O(\tilde{n}\cdot\mathtt{GCSIZE}+\mathsf{poly}(n,\kappa))$ bits via P2P channels. The broadcasts of requests also require communication of $\mathsf{poly}(n,\kappa)$ bits via broadcast channels. In total, this step requires communication of $O((n+\tilde{n})\cdot\mathtt{GCSIZE}+\mathsf{poly}(n,\kappa))$ bits via P2P channels and $\mathsf{poly}(n,\kappa)$ bits via broadcast channels.

11. **Evaluating Garbled Circuits.** This step only contains local computation and does not require any communication.

Since the protocol can only rewind once, each step only needs to be executed at most twice, so rewinding the protocol will not affect the final communication complexity. Adding up the communication cost of all the steps, the total communication complexity of our main protocol is $O(n\cdot(\mathtt{CC}(\Pi_{\mathsf{Share}})+\mathtt{CC}(\Pi_{\mathsf{Gb}}))+(n+\tilde{n})\cdot\mathtt{GCSIZE}+\mathsf{poly}(n,\kappa))$ bits via P2P channels and $W_I\cdot\mathsf{poly}(n,\kappa)$ bits via broadcast channels. Since Π_{Share}, $\Pi_{\mathsf{Gb}}(V_j)$ for all V_j are subprotocols of Π_{GLOS}, and the outputs from $\Pi_{\mathsf{Gb}}(V_j)$ for all V_j need to be sent to an evaluator in Π_{GLOS}, we have $\mathtt{CC}(\Pi_{\mathsf{Share}})+\mathtt{CC}(\Pi_{\mathsf{Gb}})+\mathtt{GCSIZE}\le\mathtt{CC}(\Pi_{\mathsf{GLOS}})$, where $\mathtt{CC}(\Pi_{\mathsf{GLOS}})$ is the communication cost of Π_{GLOS}. From [GLOS25], we have $\mathtt{CC}(\Pi_{\mathsf{GLOS}})=O(|C'|\kappa+D\cdot N^2\kappa+\mathsf{poly}(n,\tilde{n},\kappa))$. Taking $N=\Theta(n\kappa)$ and $|C'|=|C|+W_I\cdot\mathsf{poly}(n,\kappa)$, the communication complexity of our main protocol

is $O(|C|(n+\tilde{n})\kappa + D \cdot (n+\tilde{n})n^2\kappa^3 + W_I \cdot \mathsf{poly}(n,\tilde{n},\kappa))$ bits via P2P channels and $W_I \cdot \mathsf{poly}(n,\kappa)$ bits via broadcast channels. Taking $\tilde{n} = n$ to convert our protocol into the standard model, the communication complexity becomes $O(|C|n\kappa + Dn^3\kappa^3 + W_I \cdot \mathsf{poly}(n,\kappa))$ bits via P2P channels and $W_I \cdot \mathsf{poly}(n,\kappa)$ bits via broadcast channels.

Finally, we briefly analyze the round complexity. Note that all the steps of Π_{Main} only require constant rounds of communication (in Steps 4, 6, 7, 8, 9, and 10) plus:

- The executions of $\Pi_{\mathsf{Share}}, \{\Pi_{\mathsf{Gb}}(V_j)\}_{j=1}^N$ via $\mathcal{F}_{\mathsf{VP2P}}$ (in Steps 3 and 7).
- The invocations of $\mathcal{F}_{\mathsf{VSS}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{BaseROT}}$ (in Steps 1, 2, 6, and 9).
- Opening messages via $\mathcal{F}_{\mathsf{VP2P}}$ (in Steps 9 and 10).

Each round of $\Pi_{\mathsf{Share}}, \{\Pi_{\mathsf{Gb}}(V_j)\}_{j=1}^N$ requires 5 rounds in our main protocol to let all the messages sent via $\mathcal{F}_{\mathsf{VP2P}}$. Since $\Pi_{\mathsf{Share}}, \{\Pi_{\mathsf{Gb}}(V_j)\}_{j=1}^N$ are subprotocols of the constant-round protocol Π_{GLOS}, the steps of executing $\Pi_{\mathsf{Share}}, \{\Pi_{\mathsf{Gb}}(V_j)\}_{j=1}^N$ remain to be constant-round. From Lemma 2, Lemma 3, and Lemma 5, the additional number of rounds to instantiate $\mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{VSS}}, \mathcal{F}_{\mathsf{BaseROT}}$ is a constant. Moreover, opening a message via $\mathcal{F}_{\mathsf{VP2P}}$ only requires 3 rounds of communication, and there are only a constant number of rounds in Steps 9 and 10 that require opening a message. Therefore, all the steps only require a constant number of rounds. In addition, the parties at most rewind the protocol once, so there are only a constant number of steps in an execution of Π_{Main}. Thus, the total round complexity is $O(1)$.