Secure Multiparty Computation in the Presence of Covert Adaptive Adversaries

Isheeta Nargis and Anwar Hasan

University of Waterloo
Waterloo, Ontario, Canada
isheeta@gmail.com, ahasan@uwaterloo.ca

Abstract. We design a new MPC protocol for arithmetic circuits secure against erasure-free covert adaptive adversaries with deterrence $\frac{1}{2}$. The new MPC protocol has the same asymptotic communication cost, the number of PKE operations and the number of exponentiation operations as the most efficient MPC protocol for arithmetic circuits secure against covert static adversaries. That means, the new MPC protocol improves security from covert static security to covert adaptive adversary almost for free. For MPC problems where the number of parties n is much larger than the number of multiplication gates M, the new MPC protocol asymptotically improves communication complexity over the most efficient MPC protocol for arithmetic circuits secure against erasure-free active adaptive adversaries.
Keywords: Covert Adversary, Covert Adaptive Adversary, Threshold Encryption, Lossy Encryption, Public Key Encryption, Homomorphic Encryption.

1 Introduction

In a secure multiparty computation (MPC) problem, a group of mutually distrusting parties compute a possibly randomized function of their inputs in such a way that the privacy of inputs is maintained and the computed output follows the distribution of the function definition. MPC is a very strong primitive in cryptography since almost all cryptographic problems can be solved, in principle, by a general secure MPC protocol. There are many applications of secure MPC such as financial analysis, secure auction, privacy-preserving biometric identification, secure computation on gene sequences, private information retrieval, private set intersection and privacy-preserving machine learning.

The problems and risks associated with an MPC problem are modeled by an entity called the *adversary*. The adversary tries to control parties. A party which is controlled by the adversary is called a *corrupted party*. A party which is not controlled by the adversary is called an *honest party*. Depending on the assumption on the computational power of the adversary and the communication channel, there are two types of security models. In *cryptographic model of security*, it is assumed that the adversary can see all the communication between any pair of parties and the adversary is a probabilistic polynomial-time Turing machine. In *information-theoretic model of security*, it is assumed that the adversary cannot see the communication between any pair of honest parties and the computational power of the adversary is unlimited. An adversary that corrupts at most t parties is called a *t-limited adversary* or a *threshold adversary*. In that case, the number t is called the *threshold*.

Depending on the assumption of honest majority, two different security models for multiparty computation problems are defined. Let n denote the number of parties and t denote the threshold. In *multiparty model with honest majority*, it is assumed that the adversary can corrupt at most $t < \frac{n}{2}$ parties. In *multiparty model without honest majority*, it is assumed that the adversary can corrupt at most $t < n$ parties.

In order to simplify the analysis, the efficiency of protocols is generally measured in terms of a special parameter called the *security parameter*. All parties and the adversary get the security parameter as an input. The efficiency of MPC protocols are measured by some metrics. One *round* of an MPC protocol is a sequence of steps of the MPC protocol such that each party sends one message to each other party in that sequence. The *round complexity* of an MPC protocol is the number of rounds needed for executing that protocol. The *communication complexity* of an MPC protocol is the total communication (in bits) among the parties during the execution of that MPC protocol. The *computational complexity* of an MPC protocol is the asymptotic computational complexity needed for executing that protocol. Sometimes some other parameters are also used to get an estimate of the computational complexity of an MPC protocol. Many MPC protocols use public key encryption (PKE) scheme as its building blocks. Usually the PKE operations constitute the main bottleneck in the time consumed by an MPC protocol. For this reason, the number of PKE operations performed by each party gives a good measure of the computational complexity of MPC protocols. The number of exponentiation operations performed by each party is another performance metric of MPC protocols since the exponentiation operations take a big amount of time.

In *passive adversary model*, it is assumed that the corrupted parties collaborate together to learn more about the inputs and outputs of the honest parties but the corrupted parties still follow the protocol. In *active adversary model*, the corrupted parties can behave in any possible way, including the violation of the protocol. Active adversary model portrays the real world scenario better than passive adversary model. Active adversary notion is more secure than passive adversary notion. Usually the protocols for passive adversary model are simpler and more efficient than protocols for active adversary model. In *static adversary model*, it is assumed that the adversary selects the parties to corrupt before the protocol starts and the set of corrupted parties remain fixed throughout the execution of the protocol. In *adaptive adversary model*, the adversary can corrupt a party at any time, even after the the execution of the protocol is finished. Adaptive adversary model is more realistic that static adversary model. Adaptive adversary model is a stronger security model than static adversary model. It is possible to design simpler and more efficient protocols for static adversary model than the protocols for adaptive adversary model. Depending on the assumption of erasure, there are two types of adaptive adversary model. In *adaptive adversary model with erasure*, it is assumed that the parties can erase some local data. In *erasure-free adaptive adversary model*, it is assumed that the adversary learns the full history of a party when it corrupts that party. Assuming erasure is unrealistic as complete erasure is sometimes impossible to achieve. Moreover, erasure is a property that cannot be verified by another party. For these reasons, erasure-free adaptive adversary model is more realistic than adaptive adversary model with erasure. The protocols for adaptive adversary model with erasure are simpler than the protocols for erasure-free adaptive adversaries.

The *one-sided adaptive adversary model* for two-party computation (2PC) [HP14] assumes that the adversary is adaptive and it can corrupt at most one party. This is a relaxation from the *standard* or *fully adaptive adversary model for 2PC* where the adversary can corrupt both parties. Protocols for one-sided adaptive adversaries are significantly more efficient than protocols for fully adaptive adversaries [HP14,Nar17,**?**,**?**].

Aumann and Lindell [AL10] defined a new type of adversaries called the covert adversaries. In *covert adversary model* [AL10], the corrupted parties can behave in any possible way like active adversaries, but any party that attempts to cheat is guaranteed to get caught by the honest parties with a minimum fixed probability. That probability is called the *deterrence factor* of covert adversary model. This definition is suitable in many application settings including business, financial, political and diplomatic setting where getting caught cheating results in a loss of reputation, embarrassment or negative press. Security-wise, covert adversary is stronger than passive adversary and weaker than active adversary [AL10]. It is more realistic than passive adversary model. Protocols for covert adversaries are significantly more efficient than protocols for active adversaries [AL10].

Aumann and Lindell [AL10] defined covert adversary model only for static corruption. Adaptive adversary model is more realistic and more secure than static adversaries.

Nargis [NH24] defined a new adversary model, the covert adaptive adversary model, by generalizing the definition of covert adversary model for the more realistic adaptive corruption. Since the original covert adversary model defined by Aumann and Lindell is only defined for static corruption, the covert adversary model of Aumann and Lindell can be called the *covert static adverasry model.* Nargis [NH24] proved that covert adaptive security implies passive adaptive security and active adapative security implies covert adaptive security. Nargis [NH24] proved that covert adaptive security is strictly stronger than covert static security. Nargis [NH24] proved the sequential composition theorem for the new adversary model which is necessary to allow modular design of protocols for this new adversary model.

Let n denote the number of parties and M denote the number of multiplication gates in the arithmetic circuit representing the functionality to be computed. Damgård and Nielsen [DN03] designed the most efficient MPC protocol secure against erasure-free active adaptive adversaries. Their protocol needs $O(Mn^2s + n^3s)$ communication cost.

Nargis [NME13] designed an MPC protocol for arithmetic circuits secure against covert static adversaries. Her protocol is designed for the multiparty model with dishonest majority. Her protocol uses cut-and-choose techniques to additive sharing of the inputs and intermediate values, and a lossy additive homomorphic public key encryption (PKE) scheme. In all stages of the computation her protocol maintains the following invariant: each party holds an additive share, an associated randomness and an encrypted share of other parties for each wire that has been evaluated so far. Parties evaluate the multiplication gates by splitting their shares into subshares, broadcasting encryptions of subshares and performing homomorphic multiplication by their own subshares to these encryptions while a randomly selected portion of the computations are opened for verification. After evaluating all the gates in the circuit, each party sends its share and randomness calculated for the output wire of each other party. The receiving party holds the encryption of the shares of the remaining parties for this wire and uses this encryption to check the consistency of the received share and randomness. In this way, the encryption acts as a commitment to ensure that a party trying to send an invalid share gets caught. Due to the binding property of a traditional encryption scheme, a simulation-based proof of the above idea is not possible. At the end, the simulator has to generate shares and randomness on behalf of the honest parties, in a way that is consistent with the actual outputs of the corrupted parties (based on the actual inputs of the honest parties) and the messages transmitted so far (based on dummy inputs of the honest parties). This task is not possible given a traditional encryption scheme. Nargis [NME13] used lossy PKE encryption scheme to achieve that goal. In a lossy PKE scheme, a ciphertext generated using a lossy key can be opened as an encryption of any message of choice if the secret key is known. But this creates another security problem. A corrupted party can try to cheat by using a lossy key in the protocol. To prevent such an attack, a cut-and-choose verification of the key generation is also incorporated in the protocol.

The MPC protocol of [NME13] needs $O(Mn^2s)$ communication cost, $O(Mn^3)$ PKE operations and $O(Mn^3)$ exponenetiation operations for deterrence factor $\frac{1}{2}$.

We design an MPC protocol for arithmetic circuits secure against the new covert adaptive adversaries with deterrence $\frac{1}{2}$. We design the new MPC protocol by modifying the MPC protocol of [NME13] in the following way. The MPC protocol of [NME13] uses a lossy PKE scheme. The new MPC protocol uses a PKE scheme that is a combination of a lossy PKE scheme and threshold PKE scheme that is secure against erasure-free active adaptive adversaries.

The new MPC protocol needs $O(Mn^2s)$ communication cost, $O(Mn^3)$ PKE operations and $O(Mn^3)$ exponenetiation operations for deterrence factor $\frac{1}{2}$. The new MPC protocol has the same asymptotic communication cost, the number of PKE operations and the number of exponentiation operations as the MPC protocol of Nargis [NME13]. That means, the new MPC protocol improves security from covert static security to covert adaptive adversary almost for free.

For MPC problems where the number of parties n is much larger than the number of multiplication gates M, the new MPC protocol asymptotically improves communication complexity over the most efficient MPC protocol (the protocol of Damgård and Nielsen [DN03]) for arithmetic circuits secure against erasure-free active adaptive adversaries.

2 Background

2.1 Preliminary Definitions

For a set R, let $r \overset{\$}{\leftarrow} R$ denote that r is obtained by sampling uniformly at random from R. For a probabilistic polynomial-time algorithm A, let $Coins(A)$ denote the distribution of the internal randomness of A. The notation $y = A(x, r)$ means that y has been computed by running A on input x and randomness r. The notation $y \leftarrow A(x)$ means that y should be computed by running A on input x and randomness r where $r \xleftarrow{\$} Coins(A)$. For a binary string x, let $|x|$ denote the length of x.

Definition 1. *The security parameter is an additional integer valued parameter used to specify the guaranteed "level of security". All the parties and the adversary receive the security parameter as an input. The security parameter is denoted by s. All complexity characteristics (or the efficiency parameters) are measured in terms of the security parameter.*

An MPC problem is defined by a functionality.

Definition 2 ([Gol09]). *Let n denote the number of parties. An n-party functionality, denoted $f : (\{0,1\}^*)^n \rightarrow (\{0,1\}^*)^n$ is a random process that maps sequences of the form $\overline{x} = \{x_1, \ldots, x_n\}$ into sequences of random variables $f(\overline{x}) = (f_1(\overline{x}), \ldots, f_n(\overline{x}))$. For each $i \in \{1, \ldots, n\}$, the i-th party, P_i, has input x_i and wishes to obtain the i-th element in $f(x_1, \ldots, x_n)$, denoted*
$f_i(x_1, \ldots, x_n)$. Functions mapping n inputs to n outputs are a special case of functionality. Functionalities are randomized extensions of functions.

There are two types of functionalities – non-reactive functionalities and reactive functionalities.

Definition 3. *In a standard functionality or non-reactive functionality, each party has a single input and a single output. The output of each party is a probabilistic function of the inputs of all the parties. This is also called secure function evaluation which is a widely used term.*

Definition 4 ([Gol09]). *In a reactive functionality, parties perform some computations for multiple iterations. There exists a global state that is updated in each iteration. The global state may not be known by any individual party. It is shared among the parties. Initially, the global state is an empty state. Each iteration proceeds in the following way.*

1. *Each party receives an input for current iteration.*
2. *Parties compute the outputs for current iteration. The outputs of parties in current iteration depend on the inputs of the parties in current iteration and the global state in current iteration.*
3. *Parties update the global state for the next iteration based on the inputs of the parties in current iteration and the global state in current iteration.*

Interactive Turing machines are an extension of classical Turing machines in the sense that they allow interaction among the machines. For more details on their definition, see [GMR85] and [Gol06], page 191. In an MPC problem, each party is modeled by an Interactive Turing machine.

Parties use a protocol to solve an MPC problem.

Definition 5 ([Can01]). *A protocol is an algorithm written for a distributed system. A protocol describes how the parties communicate among themselves to compute a given functionality. An n-party protocol is represented as a system of n interactive Turing machines where each interactive Turing machine represents the program to be run within a different party. Conventionally, the Turing machine representing the i-th party is called P_i.*

The notion of balanced vectors are used in the security definitions of MPC protocols. A vector $\overline{x} = \{x_1, \ldots, x_n\}$ such that $x_i \in \{0,1\}^*$ for $1 \leq i \leq n$, is called *balanced*, if, for every i, j such that $1 \leq i, j, \leq n$, $|x_i| = |x_j|$.

The notion of distribution ensembles are used in the security definition of MPC protocols.

Definition 6 ([Can00]). *A distribution ensemble, $X = \{X(s,a)\}_{s \in \mathbb{N}, a \in \{0,1\}^*}$, is an infinite sequence of probability distributions, where a distribution $X(s,a)$ is associated with each value of $s \in \mathbb{N}$ and $a \in \{0,1\}^*$.*

Distribution ensembles are used to represent the outputs of computation where the parameter s denotes the security parameter and the parameter a represents the input of the computation.

The notion of computational indistinguishability is used in the security definition of MPC protocols. Intuitively, two given distribution ensembles are *computationally indistinguishable* if no efficient (polynomial-time) machine can differentiate between them.

Definition 7 ([Can00]). *Let $\delta : \mathbb{N} \to [0,1]$. Two distribution ensembles $X = \{X(s,a)\}_{s\in\mathbb{N},a\in\{0,1\}^*}$ and $Y = \{Y(s,a)\}_{s\in\mathbb{N},a\in\{0,1\}^*}$ have computational distance at most δ if, for every algorithm D that is probabilistic polynomial-time in its first input, for all sufficiently large s, all $a \in \{0,1\}^*$, and all auxiliary information $w \in \{0,1\}^*$, the following holds:*

$$|Pr[D(1^s, a, w, x) = 1] - Pr[D(1^s, a, w, y) = 1]| < \delta,$$

where x is chosen from distribution $X(s,a)$, y is chosen from distribution $Y(s,a)$, and the probabilities are taken over the choices of x, y, and the random choices of D.

Two distribution ensembles $X = \{X(s,a)\}_{s\in\mathbb{N},a\in\{0,1\}^*}$ and $Y = \{Y(s,a)\}_{s\in\mathbb{N},a\in\{0,1\}^*}$ are defined to be computationally indistinguishable, denoted $X \overset{c}{\equiv} Y$, if X and Y have computational distance at most s^{-c} for all $c > 0$.

2.2 Threshold Cryptosystem

Definition 8. ***Threshold Cryptosystem*** *[FPS00]*
A threshold cryptosystem (G, E, SD, SC) consists of the four following components.

- *A key generation algorithm G takes as input a security parameter κ, the number n of decryption servers, the threshold parameter t and a random string ω. G outputs a public key PK, a list $\{SK_1, \ldots, SK_n\}$ of private keys and a list $\{VK, VK_1, \ldots, VK_n\}$ of verification keys.*
- *An encryption algorithm E takes as input the public key PK, a plaintext m, and a random string ω. E outputs a ciphertext c.*
- *A share decryption algorithm SD takes as input the public key PK, an index i such that $1 \le i \le n$, the private key SK_i and a ciphertext c. SD outputs a decryption share c_i and a proof of its validity $proof_i$.*
- *A share combining algorithm SC takes as input the public key PK, a ciphertext c, a list $\{c_1, \ldots, c_n\}$ of decryption shares, the list $\{VK, VK_1, \ldots, VK_n\}$ of verification keys and a list $proof_1, \ldots, proof_n$ of validity proofs. SC outputs a cleartext m or a symbol $\perp$ denoting failure to decrypt.*

Semantic security of a threshold cryptosystem is defined through the game described in Figure 1.

1. The attacker chooses to corrupt t servers. She learns all their secret information and she actively controls their behavior.
2. The key generation algorithm G is invoked – then the public keys are publicized, each server receives its secret keys and the attacker learns the secrets of the corrupted players.
3. The attacker chooses a message m and a partial decryption oracle gives her valid decryption shares of the encryption of m, along with proofs of validity. This step is repeated as many times as the attacker wishes.
4. The attacker issues two messages m_0 and m_1 and sends them to an encryption oracle who randomly chooses a bit b and sends back an encryption c of M_b to the attacker.
5. The attacker repeats step 3, asking for decryption shares of encryptions of chosen messages.
6. The attacker outputs a bit b'.

Fig. 1: The game for static threshold semantic security.

Definition 9. ***Semantic Security of a Threshold Cryptosystem [FPS00]***
A threshold encryption scheme is said to be semantically secure against active static adversaries if for any polynomial time attacker, $b = b'$ with probability only negligibly greater than $1/2$.

2.3 Lossy Encryption

Bellare et al. defined *Lossy Encryption* in [BHY09], extending the definition of *Dual-Mode Encryption* of [PVW08] and *Meaningful/Meaningless Encryption* of [?]. In a lossy encryption scheme, there are two modes of operations. In the injective mode, encryption is an injective function of the plaintext. In the lossy mode, the ciphertexts generated are independent of the plaintext.

Let κ denote the security parameter. For a probabilistic polynomial time Turing machine A, let $a \overset{\$}{\leftarrow} A(x)$ denote that a is obtained by running A on input x where a is distributed according to the internal randomness of A. Let $coins(A)$ denote the distribution of the internal randomness of A. For a set R, let $r \overset{\$}{\leftarrow} R$ denote that r is obtained by sampling uniformly from R. Let $E_{pk}(m, r)$ denote the result of encryption of plaintext m using encryption key pk and randomness r. Let $D_{sk}(c)$ denote the result of decryption of ciphertext c using decryption key sk.

Definition 10. ***(Lossy Public Key Encryption Scheme [BHY09])***
A lossy public-key encryption scheme is a tuple (G, E, D) of probabilistic polynomial time algorithms such that

- *keys generated by $G(1^\kappa, inj)$ are called injective keys.*
- *keys generated by $G(1^\kappa, lossy)$ are called lossy keys.*

The algorithms must satisfy the following properties.

1. ***Correctness on injective keys.***
   For all plaintexts $x \in X$,
   $$Pr\left[(pk, sk) \overset{\$}{\leftarrow} G(1^\kappa, inj); r \overset{\$}{\leftarrow} coins(E) : D_{sk}(E_{pk}(x, r)) = x\right] = 1.$$
2. ***Indistinguishability of keys.***
   The public keys in lossy mode are computationally indistinguishable from the public keys in the injective mode.
3. ***Lossiness of lossy keys.***
   If $(pk_{lossy}, sk_{lossy}) \overset{\$}{\leftarrow} G(1^\kappa, lossy)$, then for all $x_0, x_1 \in X$, the distributions $E_{pk_{lossy}}(x_0, R)$ and $E_{pk_{lossy}}(x_1, R)$ are statistically indistinguishable.
4. ***Openability.***
   If $(pk_{lossy}, sk_{lossy}) \overset{\$}{\leftarrow} G(1^\kappa, lossy)$ and $r \overset{\$}{\leftarrow} coins(E)$, then for all $x_0, x_1 \in X$ with overwhelming probability, theres exists $r' \in coins(E)$ such that
   $$E_{pk_{lossy}}(x_0, r) = E_{pk_{lossy}}(x_1, r').$$
   That is, there exists a (possibly inefficient) algorithm opener that can open a lossy ciphertext to any arbitrary plaintext with all but negligible probability.

The semantic security of a lossy encryption scheme is implied by definition, as follows. For any $x_0, x_1 \in X$,

$$E_{proj(G(1^\kappa,inj))}(x_0, R) \overset{c}{\equiv} E_{proj(G(1^\kappa,lossy))}(x_0, R) \overset{s}{\equiv} E_{proj(G(1^\kappa,lossy))}(x_1, R) \overset{c}{\equiv} E_{proj(G(1^\kappa,inj))}(x_1, R).$$

Definition 11. ***(Key Pair Detection)***
A lossy encryption scheme (G, E, D) is said to satisfy key pair detection, if it holds that it can be decided in polynomial time whether a given pair (PK_i, SK_i) of keys generated by invoking G is a lossy pair or an injective pair.

In our protocol, we use a public key encryption scheme that satisfies the following properties.

1. additive homomorphic,
2. lossy encryption with an efficient (polynomial time) *Opener* algorithm, and
3. key pair detection.

Hemenway et al. [**?**] designed a lossy encryption scheme based on Paillier's encryption scheme. This scheme satisfies all these required properties.

2.4 Lossy Threshold Public Key Encryption Scheme

Simply stated, a lossy threshold PKE scheme is the combination of a lossy PKE scheme and a threshold PKE scheme.

Definition 12. *A lossy threshold PKE scheme secure against erasure-free t-limited active adaptive adversaries is a threshold PKE scheme*
(K, KG, E, Π_{DEC}) *, with the following modifications.*

Key Generation: *The input of the key generation algorithm KG is $(1^s, mode)$. Here, s is the security parameter, and $mode \in \{0,1\}$ denotes the mode of the key generated, for a lossy PKE scheme.*
Lossy Encryption Properties: *The encryption scheme is a lossy PKE scheme.*

2.5 The Decisional Composite Residuosity Assumption

For two non-negative integers x, y, let $gcd(x,y)$ denote the greatest common divisor of x and y. For any integer $N > 1$, define Z_N^* to be the set $\mathbb{Z}_N^* = \{a \in \{1,\ldots,N-1\} | gcd(a,N) = 1\}$. $\mathbb{Z}_N^*$ is a group with multiplication modulo N as the group operation. Let $N = pq$ be a product of two large primes p and q. First, the definition of N-th residue is presented.

Definition 13 ([Pai99]). *A number z is said to be a N-th residue modulo N^2, if there exists a number $y \in \mathbb{Z}_{N^2}^*$ such that $z = y^N \bmod N^2$.*

Next, the problem of deciding N-th residue is defined.

Definition 14 ([Pai99]). *The problem of deciding N-th residuosity is defined to be the problem of distinguishing N-th residues from the non-N-th residues.*

The decisional composite residuosity assumption is defined below.

Definition 15 ([Pai99]). *The decisional composite residuosity assumption states that there exists no polynomial time algorithm for deciding N-th residuosity.*

2.6 Security Definition of MPC Protocols Secure against Covert Adaptive Adversaries

In this section we present the security definition of MPC protocols in the presence of covert adaptive adversaries following Nargis [NH24].

The Ideal World in Covert Adaptive Adversary Model
Let $f : (\{0,1\}^*)^n \rightarrow (\{0,1\}^*)^n$ be an n-party functionality, where $f = (f_1, \ldots, f_n)$. There exists an incorruptible trusted party in the ideal world. Let $P_1, \ldots, P_n$ denote the parties. Party P_i has input $x_i \in \{0,1\}^*$ – no random input is required. Let $\overline{x} = \{x_1, \ldots, x_n\}$. Parties want to evaluate $f(\overline{x}) =$

$\{f_1(\overline{x}), \ldots, f_n(\overline{x})\}$. Let $\mathcal{A}$ denote an adaptive ideal world adversary that is a non-uniform interactive probabilistic polynomial time Turing machine with random input r_0 and security parameter s. Let $\mathcal{Z}$ denote the environment that is a non-uniform interactive probabilistic polynomial time Turing machine with input z, random input r_z and security parameter s. Let $\epsilon : \mathbb{N} \to [0, 1]$ be a function.
The execution in the ideal world with ϵ proceeds as follows.

Inputs:

Each party obtains an input. Let x_i denote the input of P_i.

First Corruption Stage:

The adversary $\mathcal{A}$ receives auxiliary information from $\mathcal{Z}$. Then $\mathcal{A}$ proceeds in iterations. In each iteration, $\mathcal{A}$ may decide to corrupt some party, depending on $\mathcal{A}$'s random input and the information collected so far. When a party is corrupted, the adversary $\mathcal{A}$ gets its input, the environment $\mathcal{Z}$ learns the identity of the corrupted party and sends some extra auxiliary information to $\mathcal{A}$. This information represents the internal history of the newly corrupted party in other protocol executions. Let I_1 denote the set of corrupted parties at the end of this stage.

Computation Stage:

Send inputs to the Trusted Party:

Each honest party P_j sends its received input x_j to the trusted party. The corrupted parties, controlled by $\mathcal{A}$, may either send their received input, or send some other input of the same length to the trusted party. This decision is made by $\mathcal{A}$ and may depend on the information gathered so far. Let $\overline{w} = \{w_1, \ldots, w_n\}$ denote the vector of inputs sent to the trusted party.

Abort Options:

If a corrupted party sends $w_i = abort_i$ to the trusted party as its input, then the trusted party sends $abort_i$ to all of the honest parties and halts. If a corrupted party sends $w_i = corrupted_i$ to the trusted party as its input, then the trusted party sends $corrupted_i$ to all of the honest parties and halts. If multiple parties send $abort_i$ (respectively, $corrupted_i$), then the trusted party relates only to one of them (say, the one with the smallest i). If both $corrupted_i$ and $abort_j$ messages are sent, then the trusted party ignores the $corrupted_i$ message.

Attempted Cheat Option:

If a corrupted party sends $w_i = cheat_i$ to the trusted party as its input, then the trusted party works as follows:

1. With probability ϵ, the trusted party sends $corrupted_i$ to the adversary and all of the honest parties.
2. With probability $(1 - \epsilon)$, the trusted party sends *undetected* to the adversary along with the honest parties' inputs $\{x_j\}_{j \in [n] \setminus I_1}$. Following this, the adversary sends the trusted party output values $\{y_j\}_{j \in [n] \setminus I_1}$ of its choice for the honest parties. Then, for every $j \in [n] \setminus I_1$, the trusted party sends y_j to P_j.

The ideal execution then ends here.

If no w_i equals $abort_i, corrupted_i$ or $cheat_i$, then the ideal execution continues below.

Trusted Party Answers Adversary:

The trusted party computes $(f_1(\overline{w}), \ldots, f_n(\overline{w}))$ and sends $f_i(\overline{w})$ to $\mathcal{A}$, for all $i \in I_1$.

Second Corruption Stage:

After learning the outputs of the corrupted parties, $\mathcal{A}$ proceeds in another sequence of iterations. In each iteration, $\mathcal{A}$ may decide to corrupt some party, where the decision of $\mathcal{A}$ depends on the information obtained so far. When a party is corrupted, the adversary $\mathcal{A}$ gets its input and output, the environment $\mathcal{Z}$ learns the identity of the corrupted party and sends some extra auxiliary information to $\mathcal{A}$. This information represents the internal history of the newly corrupted party in other protocol executions. Let I_2 denote the set of corrupted parties at the end of this stage.

Trusted Party Answers Honest Parties:

After the second corruption stage, the adversary sends either $abort_i$ for some $i \in I_2$, or *continue* to the trusted party.

If the trusted party receives $abort_i$ for some $i \in I_2$, then it sends $abort_i$ to all honest parties and halts.

If the trusted party receives *continue*, then it sends $f_j(\overline{w})$ to party P_j, for each $j \in [n] \setminus I_2$.

Outputs:

An honest party always outputs the message it obtained from the trusted party. The corrupted parties output nothing. The adversary $\mathcal{A}$ outputs any arbitrary (probabilistic polynomial-time computable) function of the information gathered during the computation in the ideal world. The environment $\mathcal{Z}$ learns all outputs.

Post-Execution Corruption Stage:

The environment $\mathcal{Z}$ and the adversary $\mathcal{A}$ interacts in rounds. In each round, $\mathcal{Z}$ generates a "Corrupt P_i" request for some P_i to $\mathcal{A}$. After receiving this request, $\mathcal{A}$ sends $\mathcal{Z}$ some arbitrary information. This information represents the internal history of P_i pertaining to the evaluation of f. For this purpose, $\mathcal{A}$ may corrupt more parties as described in the second corruption stage. The interaction continues until $\mathcal{Z}$ halts, with some output. Without loss of generality, the output of $\mathcal{Z}$ can be defined as the entire view of $\mathcal{Z}$ during its interaction with $\mathcal{A}$ and the parties. Let I_3 denote the set of corrupted parties at the end of this stage. The global output is defined to be the output of $\mathcal{Z}$. The output of $\mathcal{Z}$ may include the outputs of all parties and the output of the adversary $\mathcal{A}$.

Definition 16. *Let $IDEALCA^{\epsilon}_{f,\mathcal{A},\mathcal{Z}}(s, \overline{x}, z, \overline{r})$ denote the output of the environment $\mathcal{Z}$ after parties $P_1, \dots, P_n$ performing an evaluation of f with deterrence ϵ in the ideal world of covert adaptive adversary model in the presence of adversary $\mathcal{A}$ where party P_i has input x_i, the adversary $\mathcal{A}$ has random input r_0, the environment $\mathcal{Z}$ has input z and random input r_z, the input vector is $\overline{x} = \{x_1, \dots, x_n\}$, the vector of random inputs is $\overline{r} = \{r_z, r_0\}$, and s is the security parameter. Let $IDEALCA^{\epsilon}_{f,\mathcal{A},\mathcal{Z}}(s, \overline{x}, z)$ denote the random variable describing the distribution of $IDEALCA^{\epsilon}_{f,\mathcal{A},\mathcal{Z}}(s, \overline{x}, z, \overline{r})$ where $\overline{r}$ is selected uniformly at random from its domain.*

The Real World in Covert Adaptive Adversary Model

The real world in the covert adaptive adversary model is the same as the real world in the active adaptive adversary model. For completeness, we describe the real world here.

Let $f : (\{0,1\}^*)^n \rightarrow (\{0,1\}^*)^n$ be an n-party functionality, where $f = (f_1, \dots, f_n)$. Let Π be an n-party protocol that evaluates f. There is no trusted party. The adversary $\mathcal{A}$ sends all messages in place of the corrupted parties and may follow an arbitrary polynomial-time strategy. The honest parties follow the instructions of Π. Each party P_i has input $x_i \in \{0,1\}^*$, random input $r_i \in \{0,1\}^*$ and the security parameter s. Let $\overline{x} = \{x_1, \dots, x_n\}$. Let $\mathcal{A}$ denote an adaptive real world adversary that is a non-uniform interactive probabilistic polynomial time Turing machine with random input r_0 and security parameter s. Let $\mathcal{Z}$ denote the environment that is a non-uniform interactive probabilistic polynomial time Turing machine with input z, random input r_z and security parameter s.

At first the adversary $\mathcal{A}$ receives some auxiliary information from the environment $\mathcal{Z}$. The computation proceeds in rounds. Each round proceeds in a series of mini-rounds. At the start of each mini-round, the adversary $\mathcal{A}$ may corrupt parties one by one in an adaptive way, depending on the information gathered so far. Then $\mathcal{A}$ selects an honest party P_i that has not been activated in this round, and activates it. When activated, P_i receives the messages sent to it in the previous round and generates its messages for this round – then the mini-round ends. $\mathcal{A}$ learns all messages sent by P_i to the corrupted parties. When all the honest parties have been activated, $\mathcal{A}$ generates the messages to be sent by the corrupted parties that were not activated in this round, and the next round begins.

When a party is corrupted, $\mathcal{A}$ learns the party's input, random input, and the entire history of the messages sent and received by the party. $\mathcal{Z}$ learns the identity of the corrupted party and sends some additional auxiliary information to $\mathcal{A}$. This information represents the party's internal data from other protocol executions. From this point on $\mathcal{A}$ learns all the messages received by the party and the party behaves according to the instruction of $\mathcal{A}$.

At the end of the protocol execution, the honest parties output whatever is specified by the protocol. The corrupted parties output nothing. The adversary $\mathcal{A}$ outputs some arbitrary (probabilistic polynomial time computable) function computed from its view. The environment $\mathcal{Z}$ learns all outputs.

Then a "Post-Execution Corruption Stage" begins. In this stage, $\mathcal{Z}$ and $\mathcal{A}$ interacts in rounds. In each round, $\mathcal{Z}$ generates a "Corrupt P_i" request to $\mathcal{A}$. Then $\mathcal{A}$ sends $\mathcal{Z}$ some arbitrary information. This information represents the internal data of P_i during the execution of protocol Π. The interaction continues until $\mathcal{Z}$ halts, with some output. The output of $\mathcal{Z}$ is defined to be its entire view during its interaction with $\mathcal{A}$. The global output is defined to be the output of $\mathcal{Z}$.

Below we describe the definition of the execution in the real world of the active adaptive adversary model which we obtain by plugging in the active adversary model in the generic definition of the execution in the real world of the adaptive adversary model.

Definition 17. *Let $REAL_{\Pi,\mathcal{A},\mathcal{Z}}\left(s,\overline{x},z,\overline{r}\right)$ denote the output of the environment $\mathcal{Z}$ after parties $P_1,\ldots,P_n$ running protocol Π in the real world of active adaptive adversary model in the presence of adversary $\mathcal{A}$ where party P_i has input x_i and random input r_i, the adversary $\mathcal{A}$ has random input r_0, the environment $\mathcal{Z}$ has input z and random input r_z, the input vector is $\overline{x}=\{x_1,\ldots,x_n\}$, the vector of random inputs is $\overline{r}=\{r_z,r_0,r_1,\ldots,r_n\}$, and s is the security parameter. Let $REAL_{\Pi,\mathcal{A},\mathcal{Z}}\left(s,\overline{x},z\right)$ denote the random variable describing the distribution of $REAL_{\Pi,\mathcal{A},\mathcal{Z}}\left(s,\overline{x},z,\overline{r}\right)$ where $\overline{r}$ is selected uniformly at random from its domain.*

Security of MPC protocols in the presence of covert adaptive adversaries with deterrence ϵ is defined as follows.

Definition 18. ***(Security in the presence of Covert Adaptive Adversaries***

A protocol Π is said to securely compute f in the presence of covert adaptive adversaries with deterrence ϵ if for any non-uniform probabilistic polynomial-time adaptive adversary $\mathcal{A}$ for the real world and any environment $\mathcal{Z}$, there exists a non-uniform probabilistic polynomial-time adaptive adversary $\mathcal{S}$ for the ideal world such that the following holds for every balanced vector $\overline{x}$

$$\left\{IDEALCA^{\epsilon}_{f,\mathcal{S},\mathcal{Z}}(s,\overline{x},z)\right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}}\overset{c}{\equiv}\left\{REAL_{\Pi,\mathcal{A},\mathcal{Z}}(s,\overline{x},z)\right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}}.$$

2.7 Security Relationship of the New Covert Adaptive Adversary Model with Existing Adversary Models

In this section, we describe the security relations of the new covert adaptive adversary model with active adaptive adversary model, passive adaptive adversary model and covert static adversary model following Nargis [NH24].

Nargis [NH24] proved that active adaptive security implies covert adptive security.

Proposition 1 ([NH24]). *Let Π be a protocol that securely computes some functionality f with abort in the presence of active adaptive adversaries. Then, Π securely computes f in the presence of covert adaptive adversaries with deterrence ϵ for every ϵ such that $0\leq\epsilon\leq 1$.*

Definition 19. *A passive adaptive adversary that is allowed to modify its input before the execution of the protocol begins is called an augmented passive adaptive adversary.*

Nargis [NH24] proved that covert adaptive security implies passive adptive security.

Proposition 2. *Let Π be a protocol that securely computes some functionality f in the presence of covert adaptive adversaries with deterrence ϵ and for $\epsilon(s)\geq 1/poly(s)$. Then, Π securely computes f in the presence of augmented passive adaptive adversaries.*

We first define a transformation.

Definition 20 ([NH24]). *Let Π be a protocol that securely computes some functionality f in the presence of covert adaptive adversaries with deterrence ϵ. Let $ToActive()$ denote the transformation of Π to a protocol Π' such that if an honest party is supposed to output $corrupted_i$ in Π, then it outputs $abort_i$ in Π'.*

Proposition 3 ([NH24]). *Let Π be a protocol and $\mu(s)$ be a negligible function of s. Let $\Pi'=ToActive(\Pi)$. Then Π securely computes some functionality f in the presence of covert adaptive adversaries with deterrence $\epsilon(s)=1-\mu(s)$ if and only if Π' securely computes f with abort in the presence of active adaptive adversaries.*

Nargis [NH24] proved that covert adaptive security implies covert static security and covert adaptive security is strictly stronger than covert static security.

Proposition 4 ([NH24]). *Let Π be a protocol that securely computes functionality f in the presence of covert adaptive adversaries with deterrence ϵ. Then, Π securely computes f in the presence of covert static adversaries with deterrence ϵ. Furthermore, assuming the presence of homomorphic public key encryption schemes, there exists protocols that are secure in the presence of covert static adversaries, but not secure in the presence of covert adaptive adversaries.*

2.8 Sequential Composition Theorem for Covert Adaptive Adversaries

Nargis [NH24] proved sequential composition theorem for covert adaptive adversary model.

The Hybrid World.

We first define the hybrid world where the parties run an n-party protocol Π that contains ideal evaluations of some n-party functionalities $f_1, \dots, f_{p(s)}$. To evaluate these ideal functionalities, parties have access to a trusted party. The trusted party is called in special rounds, determined by protocol Π. In each such round, a functionality f_i (out of $f_1, \dots, f_{p(s)}$) is specified. The trusted party computing f_i acts as the trusted party in the ideal world of the covert adaptive adversary model with deterrence ϵ_i. First the adversary adaptively corrupts parties, and learns the internal data of corrupted parties. For each newly corrupted party, the adversary receives information from the environment. Then each honest party P_j sends its input for f_i to the trusted party. The input of an honest party P_j for functionality f_i depends on protocol Π. The adversary sends the inputs of the corrupted parties for f_i to the trusted party. The honest parties all send their inputs for f_i to the trusted party in the same round. The honest parties do not communicate among themselves until they receive their output of f_i from the trusted party. The reason for this is that we are considering *sequential composition* where the ideal evaluation of a functionality f_i is fully finished before the start of the next ideal evaluation of functionality f_{i+1}. The trusted party computes functionality f_i on the received inputs and then sends the respective outputs of f_i to each party. Then the adversary can again adaptively corrupt more parties. After receiving the outputs for f_i from the trusted party, the parties resume the execution of protocol Π. Such a hybrid world is called the $(f_1, \epsilon_1), \dots, (f_{p(s)}, \epsilon_{p(s)})$-hybrid world.

Definition 21. *Let $HYBRID_{\Pi,\mathcal{A},\mathcal{Z}}^{(f_1,\epsilon_1),\dots,(f_{p(s)},\epsilon_{p(s)})}(s, \overline{x}, z, \overline{r})$ denote the output of the environment $\mathcal{Z}$ after parties $P_1, \dots, P_n$ performing an execution of protocol Π in the $(f_1, \epsilon_1), \dots, (f_{p(s)}, \epsilon_{p(s)})$-hybrid world in the presence of adversary $\mathcal{A}$ where party P_i has input x_i and random input r_i, the adversary $\mathcal{A}$ has random input r_0, the environment $\mathcal{Z}$ has input z and random input r_z, the input vector is $\overline{x} = \{x_1, \dots, x_n\}$, the vector of random inputs is $\overline{r} = \{r_z, r_0, r_1, \dots, r_n\}$, and s is the security parameter. Let $HYBRID_{\Pi,\mathcal{A},\mathcal{Z}}^{(f_1,\epsilon_1),\dots,(f_{p(s)},\epsilon_{p(s)})}(s, \overline{x}, z, \overline{r})$ denote the random variable describing the distribution of $HYBRID_{\Pi,\mathcal{A},\mathcal{Z}}^{(f_1,\epsilon_1),\dots,(f_{p(s)},\epsilon_{p(s)})}(s, \overline{x}, z, \overline{r})$ where $\overline{r}$ is selected uniformly at random from its domain.*

Replacing an ideal evaluation with a subroutine call.

Now we describe how to replace an ideal evaluation of a functionality f_i with a protocol ρ_i within the execution of protocol Π. Let ℓ_i denote the round at which protocol ρ_i is invoked within protocol Π. At the start of round ℓ_i, each party P_j saves its internal state relevant to protocol Π in a special tape. Let $\sigma_{j,i}$ denote this state. The call to the trusted party for evaluation of f_i is replaced by the execution of protocol ρ_i. Let $x_{j,i}$ denote the input of P_j for functionality f_i according to protocol Π. P_j sets its input for execution of ρ_i to $x_{j,i}$ and sets its random input for ρ_i to a uniform random element of the appropriate domain. No honest party resumes execution of protocol Π before the execution of ρ_i is finished. All honest parties finish the execution of ρ_i at the same round. When P_j completes execution of protocol ρ_i with output $y_{j,i}$, P_i resumes the execution of Π starting from state $\sigma_{j,i}$ as if $y_{j,i}$ is the value that P_j received as its output for f_i from the trusted party. If P_j gets *corrupted*$_k$ as its output from the execution of ρ_i, then P_j acts as per the instruction of protocol Π. Let $\Pi^{\rho_1,\dots,\rho_{p(s)}}$ denote protocol

Π (which is initially designed for the $(f_1, \ldots, f_{p(s)}$-hybrid world) where each ideal evaluation call to f_i is replaced by a subroutine call to protocol ρ_i.

Lemma 1 **([NH24]).** *Let f be an n-party functionality. Let Π be an n-party protocol to compute f. Define h to be the reactive functionality such that h acts like the trusted party computing f with deterrence ϵ in the covert adaptive adversary model. Then Π securely computes f with deterrence ϵ in the covert adaptive adversary model if and only if Π securely computes h in the active adaptive adversary model.*

Theorem 1 **(Sequential Composition Theorem for Covert Adaptive Adversary Model.).** *[[NH24]] Let $p(s)$ be a polynomial. Let $f_1, \ldots, f_{p(s)}$ be n-party probabilistic polynomial-time functionalities. Let $\rho_1, \ldots, \rho_{p(s)}$ be protocols that securely compute functionalities $f_1, \ldots, f_{p(s)}$ in the presence of covert adaptive adversaries with deterrence $\epsilon_1, \ldots, \epsilon_{p(s)}$, respectively. Let g be an n-party functionality. Let Π be a protocol that securely computes g in the $(f_1, \epsilon_1), \ldots, (f_{p(s)}, \epsilon_{p(s)})$-hybrid world (using a single call to each f_i in such a way that no more than one ideal evaluation of a functionality is made at each round) in the presence of covert adaptive adversaries with deterrence ϵ. Then $\Pi^{\rho_1,\ldots,\rho_{p(s)}}$ securely computes g in the presence of covert adaptive adversaries with deterrence ϵ.*

3 Lossy Threshold Public Key Encryption Scheme

Nargis [Nar] designed a two-party lossy threshold PKE scheme based on the DCRA assumption secure against erasure-free one-sided active adaptive adversaries. In this section, we describe a multiparty version of the two-party lossy threshold PKE scheme of [Nar] secure againt erasure-free t-limited active adaptive adversaries.

Let R_{EQ} be the relation denoting equality of discrete logarithm. Here, the common input is (x_1, x_2, y_1, y_2), and P knows a witness $w \in \mathbb{Z}_{n\lambda}$ such that $x_1 = (y_1)^w \bmod N^2$, and $x_2 = (y_2)^w \bmod N^2$. Let QR_{N^2} denote the group of squares modulo N^2. For this relation it must hold that x_1, y_1, x_2, y_2 all are elements of the group QR_{N^2}, and y_2 is a generator of QR_{N^2}. Note that λ is unknown to all parties. Then, R_{EQ} is defined as $R_{EQ} = \left\{ \left((x_1, x_2, y_1, y_2), w\right) : x_1 \equiv (y_1)^w \bmod N^2, x_2 \equiv (y_2)^w \bmod N^2 \right\}$.

Key Generation. The key generation algorithm KG is presented below.

Algorithm KG**.**

– **Inputs:** Security parameter s, and $mode \in \{0, 1\}$. Here, $mode = 1$ denotes injective mode, and $mode = 0$ denotes lossy mode.

Outputs:

– **Public Outputs:** The public key pk, the set of verification keys $\{vk, vk_1, vk_2\}$, and a commitment key ck.
– **Secret Output of** P_i: Secret key share sk_i.

1. Select two $\frac{s}{2}$ bit primes p, q such that $p = 2p' + 1, q = 2q' + 1$, where p', q' are also primes, $N = pq$, and $gcd(N, \phi(N)) = 1$.
2. Set $\lambda = 2p'q'$. Select $\beta \xleftarrow{\$} \mathbb{Z}_N^*$.
3. Select $sk_1 \xleftarrow{\$} \mathbb{Z}_{N\lambda}$. Set $sk_2 \in \mathbb{Z}_{N\lambda}$ such that $sk_1 + sk_2 = \beta\lambda \bmod N\lambda$.
4. Select $(a, b) \xleftarrow{\$} \mathbb{Z}_N^*$. Set $g = (1 + N)^a \cdot b^N \bmod N^2$. Set $\theta = a\beta\lambda \bmod N$.
5. Select $ck \xleftarrow{\$} \mathbb{Z}_p$.
6. Select $r_q \xleftarrow{\$} \mathbb{Z}_N^*$. Set $Q = g^{mode} \cdot (r_q)^N \bmod N^2$.
7. Set $pk = (N, g, \theta, Q)$.
8. Choose a random square v from $\mathbb{Z}_{N^2}^*$. Set $vk = v$. For each $i \in \{1, 2\}$, set $vk_i = (vk)^{(sk_i)}$.
9. For each $i \in \{1, 2\}$, send $(pk, \{vk, vk_1, vk_2\}, ck, sk_i)$ to P_i as its output.

Encryption. The encryption algorithm E, on input public key $pk = (N, g, \theta, Q)$, plaintext $m \in \mathbb{Z}_N$, and randomness $r \in \mathbb{Z}_N^*$, returns $c = Q^m \cdot r^N \bmod N^2$. Since N is the product of two $\frac{s}{2}$-bit primes, N can be represented using s bits. A ciphertext $c \in \mathbb{Z}_{N^2}^*$, so the size of a ciphertext is $2s$ bits.

Threshold Decryption Protocol. The threshold decryption protocol Π_{DEC} is the same as the threshold decryption protocol of threshold Paillier PKE scheme of [LP01]. L is a function whose domain is the set $S_N = \{u < N : u \bmod N = 1\}$. The function L is defined as $L(u) = \frac{u-1}{N}$. Protocol Π_{DEC} is presented below.

Protocol Π_{DEC}.

1. Each party P_i sends $ds_i = c^{sk_i} \bmod N^2$.
2. Each party P_i proves that $\log_{c^2}\left((ds_i)^2\right) = \log_{vk}(vk_i)$, using a multiparty zero-knowledge proof for relation R_{EQ} secure against erasure-free adaptive adversaries. If P_i fails, then each other party $P_j, j \neq i$ aborts.
3. Each party P_i computes
$$m = \frac{L\left(\prod_{i\in[n]} c_i\right)}{\theta}.$$

The verification key vk_i of P_i was set to $(vk)^{sk_i}$ in Algorithm KG. Each party P_i proves that it correctly computed its decryption share by using a zero-knowledge proof for R_{EQ} that is secure against erasure-free adaptive adversaries. Damgård et al. [DJN10] designed a Σ-protocol for relation R_{EQ}. The zero-knowledge proofs in Π_{DEC} are obtained by converting the non-erasure Σ-protocol for R_{EQ} using the conversion method of Damgård [Dam00]. The proof for R_{EQ} for the encryption scheme of [Nar] requires that all the common inputs are elements of the group QR_{N^2}. For this reason, the prover P_i has to prove that $\log_{c^2}\left((ds_i)^2\right) = \log_{vk}(vk_i)$. If the computation of ds_i is correctly performed, then this relation holds.

Private Threshold Decryption to One Party P_j. The private threshold decryption protocol Π_{PDEC} is presented below.

Protocol Π_{PDEC}.

1. Each party $P_i, i \neq j$ sends $ds_i = c^{sk_i} \bmod N^2$ to P_j.
2. Each party $P_i, i \neq j$ proves to P_i that $\log_{c^2}\left((ds_i)^2\right) = \log_{vk}(vk_i)$, using a zero-knowledge proof for relation R_{EQ} secure against erasure-free adaptive adversaries. If any $P_i, i \neq j$ fails, then P_j aborts.
3. P_j computes $ds_j = c^{(sk_j)} \bmod N^2$. P_j outputs
$$m = \frac{L\left(\prod_{i\in[n]} c_i\right)}{\theta}.$$

Additive Homomorphic Properties and Blindability. Let $c_1 = E_{pk}(m_1, r_1)$ and $c_2 = E_{pk}(m_2, r_2)$ be two ciphertexts. Homomorphic addition of c_1 and c_2 is done by computing $c = c_1 \cdot c_2 \bmod N^2$. Homomorphic multiplication of a ciphertext c_1 by a plaintext m_2 is done by computing $c_2 = (c_1)^{m_2} \bmod N^2$. The *Blind* algorithm, on input ciphertext $c_1 = E_{pk}(m_1, r_1)$ works as follows:
It selects $r \overset{\$}{\leftarrow} \mathbb{Z}_N^*$, then returns $c_3 = c_1 \cdot (r)^N \bmod N^2$.

4 New MPC Protocol Secure Against Covert Adaptive Adversaries

In this section we present the new MPC protocol secure against erasure-free covert adaptive adversaries. The new MPC protocol is designed for the multiparty model with honest majority.

We design the new MPC protocol by modifying the MPC protocol of [NME13] in the following way. The MPC protocol of [NME13] uses a lossy PKE scheme. The new MPC protocol uses a PKE scheme that is a combination of a lossy PKE scheme and threshold PKE scheme that is secure against erasure-free active adaptive adversaries. The new MPC protocol has the same asymptotic communication cost and number of exponentiation operations as the MPC protocol of Nargis [NME13]. That means, the new MPC protocol improves security from covert static security to covert adaptive adversary almost for free.

Let w_k be a wire in C. Let $S_{k,i}$ denote the share of P_i for the wire w_k. Let $rs_{k,i}$ denote the randomness of P_i for the wire w_k. For each $j \in [n] \setminus \{i\}$, let $ES_{k,i,j}$ denote the encrypted share of P_j that P_i holds for the wire w_k.

We designed a protocol *Circuit* for computing functionality f in the presence of covert adaptive adversaries. The main stages of protocol *Circuit* are presented in Fig. 2. Each stage is presented in a separate figure later. Unless otherwise specified, we describe the action of each party $P_i, i \in [n]$, in the protocol.

Protocol Circuit.

Common Inputs:

1. An arithmetic circuit C describing $f : \mathbb{F}^n \to \mathbb{F}^n = \{f_1, \ldots, f_n\}$,
2. A topological ordering $(g_1, \ldots, g_\theta)$ of gates in C, and
3. An ordering $(w_1, \ldots, w_\rho)$ of wires in C such that the input wires of each gate in C has smaller indices than the index of its output wire.

Input of $P_i : x_i \in \mathbb{F}$.

Output of $P_i : y_i = f_i(x_1, \ldots, x_n) \in \mathbb{F}$.

In some steps of the protocol, each party is supposed to broadcast some message. If some party P_j does not broadcast any message in one of these steps, then P_i aborts.

1. **CRS Generation Stage.**
   Parties generate a common reference string.
2. **Key Generation Stage.**
   Parties generate keys for the lossy threshold PKE scheme.
3. **Input Sharing Stage.**
   Each party distributes additive shares of its input and the encryptions of these shares to other parties.
4. **Computation Stage.**
   Parties evaluate the circuit gate-by-gate.
5. **Output Generation Stage.**
   Parties evaluate their outputs.

Fig. 2: Protocol *Circuit*.

Protocol *Circuit* works in stages. Next we present the stages of *Circuit* and give an idea how it works

In the *CRS generation stage* (see Fig. 3), parties generate a common reference string σ. σ is used as the common reference string in commitment and opening subprotocols used during the rest of the protocol.

Parties generate a common reference string σ of length $p_1(s)$ using the protocol *CoinFlipPublic*. Here $p_1(\kappa)$ is a polynomial of κ.

Fig. 3: The CRS generation stage.

In the *key generation stage* (see Fig. 4), parties generate a key pair (PK, SK) for the threshold public-key encryption scheme.

In the *input sharing stage* (see Fig. 5), each party distributes two sets of additive shares of its input and their encryptions to other parties. One set is randomly selected for verification and the unopened shares are used in the computation. [1] Then we say that the input wires $w_1, \ldots, w_n$ of C have been *evaluated*. By saying

[1] This verification is only making sure that the encryptions of the shares are done correctly.

that a wire w_k has been evaluated we mean that the share $S_{k,i}$, randomness $rs_{k,i}$ of each party P_i for w_k and the encrypted share $ES_{k,i,j}$ that each party P_i holds for each other party P_j for w_k, have been fixed.[2]

1. **Parties generate challenge.**
   Parties generate a challenge $m_{key} \in \{1, 2\}$ using the protocol $CommittedCoinFlipPublic_\sigma$ in the CRS model.
2. **Parties broadcast injective public keys.**
   P_i generates two pairs of keys in injective mode, that is, P_i generates $(U_{i,j}, V_{i,j}) = G(1^\kappa, inj)$, for each $j \in \{1, 2\}$.
   P_i broadcasts the public keys $U_{i,1}$ and $U_{i,2}$.
3. **Parties open the challenge.**
   Parties open the challenge m_{key}, using the protocol $OpenCom_\sigma$ in the CRS model.
4. **Parties respond to challenge.**
   P_i broadcasts the private key $V_{i,m_{key}}$.
5. **Parties verify the responses.**
   For each $j \in [n] \setminus \{i\}$, P_i verifies that the key pair $(U_{j,m_{key}}, V_{j,m_{key}})$ is a valid injective pair of keys for the lossy encryption scheme being used. If this is not the case, then P_i broadcasts $corrupted_j$ and aborts.
6. **Parties fix their keys.**
   P_i performs the following steps.
   (a) P_i sets (PK_i, SK_i) to $(U_{i,3-m_{key}}, V_{i,3-m_{key}})$,
   (b) For each $j \in [n] \setminus \{i\}, P_i$ sets PK_j to $U_{j,3-m_{key}}$.

Fig. 4: The key generation stage.

In the *computation stage* (see Fig. 6), parties evaluate the circuit gate-by-gate, in the order $(g_1, \ldots, g_\theta)$. When the evaluation of gate g_δ is finished, we say that the output wire w_{z_δ} of g_δ has been evaluated. At each point of computation the following holds for each wire w_k of C that has been evaluated so far: each party P_i holds an additive share $S_{k,i}$, an associated randomness $rs_{k,i}$ and an encrypted share $ES_{k,i,j}$ of each other party P_j for the wire w_k.

If g_δ is an addition gate, each party P_i sets its share $S_{z_\delta,i}$ for the output wire of g_δ to the sum of the shares of P_i for the input wires of g_δ. Each party P_i computes its randomness $rs_{z_{\delta,i}}$ for the output wire of g_δ such that $E_{PK_i}(S_{z_{\delta,i}}, rs_{z_{\delta,i}})$ equals the result of homomorphic addition of the ciphertexts ($ES_{u_\delta,j,i}$ and $ES_{v_\delta,j,i}$, $j \neq i$) that other parties hold for the input wires of g_δ for P_i. Each party P_i computes the encrypted share $ES_{z_{\delta,i,j}}$ of each other party P_j for the output wire of g_δ locally, by performing homomorphic addition of the encrypted shares of P_j that P_i holds for the input wires of g_δ.

If g_δ is a multiplication-by-constant gate, each party P_i sets its share $S_{z_\delta,i}$ for the output wire of g_δ to the product of the share of P_i for the input wire of g_δ and q_δ where q_δ is the known constant multiplicand for gate g_δ. This constant q_δ is part of the description of C and known to all parties. Each party P_i computes its randomness $rs_{z_{\delta,i}}$ for the output wire of g_δ such that $E_{PK_i}(S_{z_{\delta,i}}, rs_{z_{\delta,i}})$ equals the result of homomorphic addition of the ciphertexts ($ES_{u_\delta,j,i}$ and $ES_{v_\delta,j,i}$, $j \neq i$) that other parties hold for the input wires of g_δ for P_i. Each party P_i computes the encrypted share $ES_{z_{\delta,i,j}}$ of each other party P_j for the output wire of g_δ locally, by performing homomorphic multiplication by q_δ to the encrypted share of P_j that P_i holds for the input wire of g_δ.

If g_δ is a multiplication gate, then each party P_i first generates two sets of random shares and broadcasts their encryptions. One set is randomly selected for verification and the unopened set will be used as the set $\{C_{i,j}\}_{j \in [n] \setminus \{i\}}$ during the evaluation of g_δ. Each party P_i splits its shares $A_i = S_{u_\delta,i}$ and $B_i = S_{v_\delta,i}$ (for the input wires of g_δ) into two additive subshares ($A_{i,1} + A_{i,2} = A_i$ and $B_{i,1} + B_{i,2} = B_i$), then broadcasts the encryptions of these subshares. Each party P_i splits its random share $C_{i,j}$ into four additive subshares ($H_{i,j,1,1}, H_{i,j,1,2}, H_{i,j,2,1}$, and $H_{i,j,2,2}$) and broadcasts their encryptions, for each $j \in [n] \setminus \{i\}$.

[2] $ES_{k,i,j}$ is supposed to be $E_{PK_j}(S_{k,j}, rs_{k,j})$.

1. **Parties generate challenge.**
   Parties generate a challenge $m_{in} \in \{1, 2\}$ using the protocol $CommittedCoinFlipPublic_\sigma$ in the CRS model.
2. **Parties broadcast encrypted shares.**
   P_i randomly selects two sets of shares $\{B_{1,i,j}\}_{j\in[n]}$ and $\{B_{2,i,j}\}_{j\in[n]}$ such that
   $$\sum_{j\in[n]} B_{1,i,j} = \sum_{j\in[n]} B_{2,i,j} = x_i.$$
   P_i randomly selects two sets of strings $\{b_{1,i,j}\}_{j\in[n]}$ and $\{b_{2,i,j}\}_{j\in[n]}$.
   For each $\ell \in \{1, 2\}$ and each $j \in [n]$, P_i broadcasts
   $$Y_{\ell,i,j} = E_{PK}(B_{\ell,i,j}, b_{\ell,i,j}).$$
3. **Parties send share and randomness to the designated parties.**
   For each $j \in [n] \setminus \{i\}$, P_i sends $\{B_{1,i,j}, B_{2,i,j}, b_{1,i,j}, b_{2,i,j}\}$ to P_j.
4. **Parties open the challenge.**
   Parties open the challenge m_{in} using the protocol $OpenCom_\sigma$ in the CRS model.
5. **Parties respond to challenge.**
   P_i broadcasts the sets $\{B_{m_{in},i,j}\}_{j\in[n]\setminus\{i\}}$ and $\{b_{m_{in},i,j}\}_{j\in[n]\setminus\{i\}}$.
6. **Parties verify the responses.**
   For each $j \in [n] \setminus \{i\}$ and each $k \in [n] \setminus \{j\}$, P_i verifies that
   $$Y_{m_{in},j,k} = E_{PK}(B_{m_{in},j,k}, b_{m_{in},j,k}).$$
   If any of the equalities does not hold, then P_i broadcasts $corrupted_j$ and aborts.
7. **Parties fix their shares, randomness and encrypted shares of other parties.** For each $k \in [n], P_i$ sets the followings for the input wire w_k of C.
   (a) P_i sets $S_{k,i}$ to $B_{3-m_{in},k,i}$,
   (b) P_i sets $rs_{k,i}$ to $b_{3-m_{in},k,i}$, and
   (c) For each $j \in [n] \setminus \{i\}, P_i$ sets $ES_{k,i,j}$ to $Y_{3-m_{in},k,j}$.

Fig. 5: The input sharing stage.

Each party P_i performs homomorphic multiplication by its own subshare $A_{i,k}$ to the encryption $Y_{i,\ell}$ of its own subshare $B_{i,\ell}$, then adds a random encryption of zero and broadcasts the resulting ciphertext $L_{i,k,\ell}$, for each $k, \ell \in \{1,2\}^2$. Each party P_i performs homomorphic multiplication by its own subshare $A_{i,k}$ to the encryption $Y_{j,\ell}$ of the subshare $B_{j,\ell}$ of each other party P_j, then adds a random ciphertext of $H_{i,j,k,\ell}$ and broadcasts the resulting ciphertext $K_{i,j,k,\ell}$, for each $k, \ell \in \{1,2\}^2$. After receiving the results of these calculations from other parties, each party P_i decrypts the ciphertexts $\{K_{j,i,k,\ell}\}_{j\in[n]\setminus\{i\},k,\ell\in\{1,2\}^2}$ under its own key PK_i, sums the results of decryptions up, then subtracts its own randomness $(\{C_{i,j}\}_{j\in[n]\setminus\{i\}})$ to get its share $S_{z_\delta,i}$ of the product. Each party P_i sets its randomness $rs_{z_\delta,i}$ for the output wire of g_δ to a string such that encrypting $S_{z_\delta,i}$ under PK_i using this string as randomness would result in the ciphertext $(ES_{z_\delta,j,i}, j \neq i)$ that the other parties would hold as the encryption of the share of P_i for the output wire of g_δ. Each party P_i computes the encryption $ES_{z_\delta,i,j}$ of the share of each other party P_j for the output wire of g_δ, by performing the corresponding homomorphic additions to the corresponding ciphertexts as all the ciphertexts (including the results after calculations) are available to all parties. Exactly half of all the calculations (splitting into subshares, encryption of subshares, homomorphic multiplication by own subshares to own encrypted subshares, and homomorphic multiplication by own subshares to other parties' encrypted subshares) are randomly selected for verification, ensuring that a party attempting to cheat gets caught with probability at least $\frac{1}{2}$. It is also verified that the homomorphic encryptions of the additive subshares (e.g. the encryptions $X_{i,1}$ and $X_{i,2}$ of subshares $A_{i,1}$ and $A_{i,2}$) and an encryption of zero (e.g. $E_{PK_i}(0, a_{i,0})$) results in the encryption of the original share (e.g. $EA_{j,i} = ES_{u_\delta,j,i}, j \neq i$, the encryption of A_i that P_j holds, that is, the encryption of the share of P_i that P_j holds for the input wire u_δ of g_δ) – the splitting party P_i has to broadcast the string to be used as randomness to encrypt zero (e.g. $a_{i,0}$) to prove this equality. A party attempting to cheat gets caught with probability 1 in this case.

In the *output generation stage* (see Fig. 7), each party P_i sends its share $S_{\gamma+k,i}$ and randomness $rs_{\gamma+k,i}$ for the output wire $w_{\gamma+k}$ to each other party P_k. The receiving party P_k holds the encryption $ES_{\gamma+k,k,i}$ of each other party P_i for its output wire $w_{\gamma+k,i}$. The receiving party P_k checks the consistency of the received input and randomness with the corresponding ciphertexts (P_k checks whether $ES_{\gamma+k,k,i} = E_{PK_i}(S_{\gamma+k,i}, rs_{\gamma+k,i})$ or not).

5 Correctness of protocol *Circuit*

We can prove correctness of the protocol *Circuit* by induction on the wires of the circuit C. The induction claim is that the shares of each wire sum up to the correct value of the wire.

We prove the base case on the input wires of C. If all parties send correct encryption of their shares in the input sharing stage, then the following holds for each $k \in [n]$:
The value x_k of the input wire w_k is shared as $(S_{k,1}, \ldots, S_{k,n})$ where $S_{k,1}, \ldots, S_{k,n} \in \mathbb{F}$, party P_j holds the share $S_{k,j}$ and

$$\sum_{j\in[n]} S_{k,j} = x_k.$$

Let $\delta \in [\theta]$. For the induction case, we consider the computation of the gate g_δ. Let H_{u_δ} and H_{v_δ} denote the correct value of wires w_{u_δ} and w_{v_δ}, respectively. Then,

$$\sum_{i\in[n]} S_{u_\delta,i} = H_{u_\delta}$$

and

$$\sum_{i\in[n]} S_{v_\delta,i} = H_{v_\delta}.$$

Next we describe the situation depending on the type of gate g_δ.

Case 1: g_δ is an addition gate.
In this case, each party $P_i, i \in [n]$, sets

$$S_{z_\delta,i} = S_{u_\delta,i} + S_{v_\delta,i}.$$

For each $\delta \in [\theta]$, parties perform the following actions, depending on the type of gate g_δ.

Case 1: g_δ is an addition gate.

1. P_i sets
$$S_{z_\delta,i} = S_{u_\delta,i} + S_{v_\delta,i}.$$
2. P_i computes $rs_{z_\delta,i}$ such that the following equality holds.
$$E_{PK}(S_{z_\delta,i}, rs_{z_\delta,i}) = E_{PK}(S_{u_\delta,i}, ra_{u_\delta,i}) +_h E_{PK}(S_{v_\delta,i}, ra_{v_\delta,i}).$$
3. For each $j \in [n] \setminus \{i\}$, P_i sets
$$ES_{z_\delta,i,j} = ES_{u_\delta,i,j} +_h ES_{v_\delta,i,j}.$$

Case 2: g_δ is a multiplication-by-constant gate.

Let $q_\delta \in \mathbb{F}$ be the constant with which the multiplication will be done.

1. P_i sets
$$S_{z_\delta,i} = q_\delta \cdot S_{u_\delta,i}.$$
2. P_i computes $rs_{z_\delta,i}$ such that the following equality holds.
$$E_{PK}(S_{z_\delta,i}, rs_{z_\delta,i}) = q_\delta \times_h E_{PK}(S_{u_\delta,i}, rs_{u_\delta,i}).$$
3. For each $j \in [n] \setminus \{i\}$, P_i sets
$$ES_{z_\delta,i,j} = q_\delta \times_h ES_{u_\delta,i,j}.$$

Case 3: g_δ is a multiplication gate.

For each $i \in [n]$, let $A_i^{(\delta)}, B_i^{(\delta)}, ra_i^{(\delta)}$ and $rb_i^{(\delta)}$ denote $S_{u_\delta,i}, S_{v_\delta,i}, rs_{u_\delta,i}$, and $rs_{v_\delta,i}$, respectively. For each $i \in [n]$ and each $j \in [n] \setminus \{i\}$, let $EA_{i,j}^{(\delta)}$ and $EB_{i,j}^{(\delta)}$ denote $ES_{u_\delta,i,j}$ and $ES_{v_\delta,i,j}$, respectively.

1. **Parties generate random shares.**
   (a) **Parties generate challenge.**
   Parties generate challenge $m_r \in \{1,2\}$ using the protocol $CommittedCoinFlipPublic_\sigma$ in the CRS model.
   (b) **Parties generate random shares.**
   P_i randomly selects two sets of shares $\{Q_{1,i,j}\}_{j\in[n]\setminus\{i\}}$ and $\{Q_{2,i,j}\}_{j\in[n]\setminus\{i\}}$ and two sets of strings $\{rq_{1,i,j}\}_{j\in[n]\setminus\{i\}}$ and $\{rq_{2,i,j}\}_{j\in[n]\setminus\{i\}}$.
   (c) **Parties broadcast encrypted shares.**
   For each $\ell \in \{1,2\}$ and each $j \in [n] \setminus \{i\}$, P_i broadcasts
$$Y_{\ell,i,j} = E_{PK}(Q_{\ell,i,j}, rq_{\ell,i,j}).$$
   (d) **Parties open the challenge.**
   Parties open the challenge m_r using the protocol $OpenCom_\sigma$ in the CRS model.
   (e) **Parties respond to challenge.**
   P_i broadcasts $Q_{m_r,i,j}$ and $rq_{m_r,i,j}$ for each $j \in [n] \setminus \{i\}$.

Fig. 6: The computation stage.

For each $\delta \in [\theta]$, parties perform the following actions, depending on the type of gate g_δ.

Case 3: g_δ is a multiplication gate.

1. **Parties generate random shares.**
   (f) **Parties verify the responses.**
   For each $j \in [n] \setminus \{i\}$ and each $k \in [n] \setminus \{j\}$, P_i verifies that
   $$Y_{m_r,j,k} = E_{PK}(Q_{m_r,j,k}, rq_{m_r,j,k}).$$
   If any of these equalities does not hold for party P_j, then P_i broadcasts *corrupted*$_j$ and aborts.
   (g) **Parties fix their randomness.**
   P_i performs the following steps.
      i. For each $j \in [n] \setminus \{i\}$, P_i sets $C^{(\delta)}_{i,j}$ to $Q_{3-m_r,i,j}$ and $rc^{(\delta)}_{i,j}$ to $rq_{3-m_r,i,j}$.
      ii. For each $j \in [n] \setminus \{i\}$ and each $k \in [n] \setminus \{j\}$, P_i sets $EC^{(\delta)}_{i,j,k}$ to $Y_{3-m_r,j,k}$.
2. **Parties generate challenge.**
   Parties generate a challenge $m^{(\delta)} \in \{1,2\}$ using the protocol *CommittedCoinFlipPublic*$_\sigma$ in the CRS model.
3. **Parties split their shares into subshares.**
   (a) P_i chooses $A^{(\delta)}_{i,1}$ and $B^{(\delta)}_{i,1}$ uniformly at random from $\mathbb{F}$.
   (b) P_i sets
   $$A^{(\delta)}_{i,2} = A^{(\delta)}_{i} - A^{(\delta)}_{i,1}, \text{ and}$$
   $$B^{(\delta)}_{i,2} = B^{(\delta)}_{i} - B^{(\delta)}_{i,1}.$$
   (c) P_i generates four random strings $a^{(\delta)}_{i,1}, a^{(\delta)}_{i,2}, b^{(\delta)}_{i,1}$ and $b^{(\delta)}_{i,2}$.
   (d) For each $j \in \{1,2\}$, P_i broadcasts
   $$X^{(\delta)}_{i,j} = E_{PK}\left(A^{(\delta)}_{i,j}, a^{(\delta)}_{i,j}\right), \text{ and}$$
   $$Y^{(\delta)}_{i,j} = E_{PK}\left(B^{(\delta)}_{i,j}, b^{(\delta)}_{i,j}\right).$$
   (e) For each $j \in [n] \setminus \{i\}$, and each $k, \ell \in \{1,2\}^2$, P_i chooses $H^{(\delta)}_{i,j,k,\ell}$ uniformly at random from $\mathbb{F}$ such that
   $$\sum_{k,\ell \in \{1,2\}^2} H^{(\delta)}_{i,j,k,\ell} = C^{(\delta)}_{i,j}.$$
   (f) For each $j \in [n] \setminus \{i\}$, and each $k, \ell \in \{1,2\}^2$, P_i chooses a random string $h^{(\delta)}_{i,j,k,\ell}$ and broadcasts
   $$G^{(\delta)}_{i,j,k,\ell} = E_{PK}\left(H^{(\delta)}_{i,j,k,\ell}, h^{(\delta)}_{i,j,k,\ell}\right).$$

Fig. 6: The computation stage (continued from previous page).

Case 3: g_δ is a multiplication gate.

4. **Parties prove their sums.**
   (a) P_i computes two strings $aa_{i,0}^{(\delta)}$ and $bb_{i,0}^{(\delta)}$ such that

$$E_{PK}\left(0, aa_{i,0}^{(\delta)}\right) = E_{PK}\left(A_i^{(\delta)}, ra_i^{(\delta)}\right) -_h X_{i,1}^{(\delta)} -_h X_{i,2}^{(\delta)}, \text{ and}$$
$$E_{PK}\left(0, bb_{i,0}^{(\delta)}\right) = E_{PK}\left(B_i^{(\delta)}, rb_i^{(\delta)}\right) -_h Y_{i,1}^{(\delta)} -_h Y_{i,2}^{(\delta)}.$$

   (b) P_i broadcasts $aa_{i,0}^{(\delta)}$ and $bb_{i,0}^{(\delta)}$.
   (c) For each $j \in [n] \setminus \{i\}$, P_i performs the following two actions.
      i. P_i computes a string $cc_{i,j,0}^{(\delta)}$ such that

$$E_{PK}\left(0, cc_{i,j,0}^{(\delta)}\right) = E_{PK}\left(C_{i,j}^{(\delta)}, rc_{i,j}^{(\delta)}\right) -_h G_{i,j,1,1}^{(\delta)} -_h G_{i,j,1,2}^{(\delta)} -_h G_{i,j,2,1}^{(\delta)} -_h G_{i,j,2,2}^{(\delta)}.$$

      ii. P_i broadcasts $cc_{i,j,0}^{(\delta)}$.
5. **Parties send output parts that depend only on their own subshares.**
   For each $k, \ell \in \{1,2\}^2$, P_i selects a random string $vv_{i,k,\ell}^{(\delta)}$, then broadcasts

$$L_{i,k,\ell}^{(\delta)} = A_{i,k}^{(\delta)} \times_h Y_{i,\ell}^{(\delta)} +_h E_{PK}\left(0, vv_{i,k,\ell}^{(\delta)}\right).$$

6. **Parties perform computations on other parties' encrypted subshares.**
   For each $k, \ell \in \{1,2\}^2$ and each $j \in [n] \setminus \{i\}$, P_i selects a random string $hh_{i,j,k,\ell}^{(\delta)}$, performs the following computation on the encrypted inputs of P_j, then broadcasts

$$K_{i,j,k,\ell}^{(\delta)} = A_{i,k}^{(\delta)} \times_h Y_{j,\ell}^{(\delta)} +_h E_{PK}\left(H_{i,j,k,\ell}^{(\delta)}, hh_{i,j,k,\ell}^{(\delta)}\right).$$

7. **Parties open the challenge.**
   Parties open the challenge $m^{(\delta)}$ using the protocol $OpenCom_\sigma$ in the CRS model.
8. **Parties respond to challenge.**
   (a) P_i broadcasts $A_{i,m^{(\delta)}}^{(\delta)}, B_{i,m^{(\delta)}}^{(\delta)}, a_{i,m^{(\delta)}}^{(\delta)}, b_{i,m^{(\delta)}}^{(\delta)}, vv_{i,m^{(\delta)},1}^{(\delta)}$ and $vv_{i,m^{(\delta)},2}^{(\delta)}$.
   (b) For each $j \in [n] \setminus \{i\}$, and each $\ell \in \{1,2\}$, P_i broadcasts $H_{i,j,m^{(\delta)},\ell}^{(\delta)}, h_{i,j,m^{(\delta)},\ell}^{(\delta)}$ and $hh_{i,j,m^{(\delta)},\ell}^{(\delta)}$.
9. **Parties verify the responses.**
   P_i verifies the following for each party $P_j, j \in [n] \setminus \{i\}$:
   (a) **P_j's encryption sums.**
      i.

$$E_{PK}\left(0, aa_{j,0}^{(\delta)}\right) = EA_{i,j}^{(\delta)} -_h X_{j,1}^{(\delta)} -_h X_{j,2}^{(\delta)}.$$

      ii.

$$E_{PK}\left(0, bb_{j,0}^{(\delta)}\right) = EB_{i,j}^{(\delta)} -_h Y_{j,1}^{(\delta)} -_h Y_{j,2}^{(\delta)}.$$

      iii. For each $k \in [n] \setminus \{j\}$,

$$E_{PK}\left(0, cc_{j,k,0}^{(\delta)}\right) = EC_{i,j,k}^{(\delta)} -_h G_{j,k,1,1}^{(\delta)} -_h G_{j,k,1,2}^{(\delta)} -_h G_{j,k,2,1}^{(\delta)} -_h G_{j,k,2,2}^{(\delta)}.$$

Fig. 6: The computation stage (continued from previous page).

Case 3: g_δ is a multiplication gate.

9. **Parties verify the responses.**
   P_i verifies the following for each party $P_j, j \in [n] \setminus \{i\}$:
   (b) P_j **knows its encrypted data.**
      i. $$X^{(\delta)}_{j,m^{(\delta)}} = E_{PK}\left(A^{(\delta)}_{j,m^{(\delta)}}, a^{(\delta)}_{j,m^{(\delta)}}\right).$$
      ii. $$Y^{(\delta)}_{j,m^{(\delta)}} = E_{PK}\left(B^{(\delta)}_{j,m^{(\delta)}}, b^{(\delta)}_{j,m^{(\delta)}}\right).$$
      iii. For each $k \in [n] \setminus \{j\}$ and each $\ell \in \{1, 2\}$,
      $$G^{(\delta)}_{j,k,m^{(\delta)},\ell} = E_{PK}\left(H^{(\delta)}_{j,k,m^{(\delta)},\ell}, h^{(\delta)}_{j,k,m^{(\delta)},\ell}\right).$$
   (c) **The computations performed by P_j on its own subshares are correct.**
      For each $\ell \in \{1, 2\}$,
      $$L^{(\delta)}_{j,m^{(\delta)},\ell} = A^{(\delta)}_{j,m^{(\delta)}} \times_h Y^{(\delta)}_{j,\ell} +_h E_{PK}\left(0, vv^{(\delta)}_{j,m^{(\delta)},\ell}\right).$$
   (d) **The computations performed by P_j on other parties' subshares are correct.**
      For each $k \in [n] \setminus \{j\}$, and for each $\ell \in \{1, 2\}$,
      $$K^{(\delta)}_{j,k,m^{(\delta)},\ell} = A^{(\delta)}_{j,m^{(\delta)}} \times_h Y^{(\delta)}_{k,\ell} +_h E_{PK}\left(H^{(\delta)}_{j,k,m^{(\delta)},\ell}, hh^{(\delta)}_{j,k,m^{(\delta)},\ell}\right).$$

   If P_j fails in any of the verifications, then P_i broadcasts *corrupted*$_j$ and aborts.
10. **Parties compute encryption of the shares of other parties.**
   For each $j \in [n] \setminus \{i\}, P_i$ computes the encrypted share $ES_{z_\delta,i,j}$ of P_j as follows.
   $$ES_{z_\delta,i,j} = \sum_{k,\ell} L^{(\delta)}_{j,k,\ell} +_h \sum_{k\in[n]\setminus\{j\}} \sum_{\ell_1,\ell_2\in\{1,2\}^2} K^{(\delta)}_{k,j,\ell_1,\ell_2} -_h \sum_{k\in[n]\setminus\{j\}} EC^{(\delta)}_{i,j,k}.$$

Fig. 6: The computation stage (continued from previous page).

Case 3: g_δ **is a multiplication gate.**

11. **Parties decrypt shares to the corresponding parties.**
    For each $j \in [n]$, parties perform the following steps.
    (a) For each $i \in [n] \setminus \{j\}$, the following actions take place.
        i. P_i **sends decryptions share to** P_j**.**
           P_i sends
$$c_{i,j} = (ES_{z_\delta,i,j})^{s_i} \bmod N^2$$
           to P_j.
        ii. P_i **proves validity of decryptions share.**
           P_i proves to P_j through a Σ-protocol that the discrete log of $(c_{i,j})^2$ in base $(ES_{z_\delta,i,j})^2$ is equal to the discrete log of v_i in base v.
        iii. **Parties reconstruct secret shares of misbehaving parties.**
           If P_i fails in the above proof, then the following actions take place.
           A. P_j broadcasts $corrupted_i$.
           B. Each party $P_k, k \in [n] \setminus \{i, j\}$, broadcasts $f_k(i)$ and $r_{k,i}$.
           C. P_j sets $Good_j = \emptyset$.
           D. For each $k \in [n] \setminus \{i\}, P_j$ performs the following steps. P_j checks whether
$$w_{k,i} = Commit_{k_i}\left(f_k(i), r_{k,i}\right)$$
              or not. If this equality holds, then P_j sets
$$Good_j = Good_j \cup \{P_k\}.$$
           E. P_j computes Δs_i by interpolating $\{f_k(i)\}_{P_k \in Good_j}$ and then computes s_i.
           F. P_j computes
$$c_{i,j} = (ES_{z_\delta,i,j})^{s_i} \bmod N^2.$$
    (b) P_j **computes its share.**
        P_j computes
$$S_{z_\delta,j} = \frac{L\left(\prod_{i\in[n]} c_{i,j}\right)}{\theta}.$$

12. **Parties compute their randomness.**
    P_i computes its randomness $rs_{z_\delta,i}$ such that the following equality holds.
$$E_{PK}\left(S_{z_\delta,i}, rs_{z_\delta,i}\right)$$
$$= \sum_{k,\ell} L^{(\delta)}_{i,k,\ell} +_h \sum_{k\in[n]\setminus\{i\}} \sum_{\ell_1,\ell_2\in\{1,2\}^2} K^{(\delta)}_{k,i,\ell_1,\ell_2} -_h \sum_{k\in[n]\setminus\{i\}} E_{PK}\left(C^{(\delta)}_{i,k}, rc^{(\delta)}_{i,k}\right).$$

Fig. 6: The computation stage (continued from previous page).

Note that the wire $w_{\gamma+k}$ is supposed to carry the output y_k of party P_k.

1. For each $k \in [n]$, the parties perform the following steps.
   (a) **Parties send their shares and randomness for the wire $w_{\gamma+k}$ to P_k.**
   $P_i, i \in [n] \setminus \{k\}$, sends $S_{\gamma+k,i}$ and $rs_{\gamma+k,i}$ to P_k.
   If some P_j does not send these to P_k in this step, then P_k aborts.
   (b) **P_k verifies the responses.**
   For each $i \in [n] \setminus \{k\}$, P_k compares $E_{PK}(S_{\gamma+k,i}, rs_{\gamma+k,i})$ with the ciphertext $ES_{\gamma+k,k,i}$ that P_k holds as the encryption of the share of P_i for the wire $w_{\gamma+k}$.
   If the ciphertexts do not match, then P_k broadcasts $corrupted_i$ and aborts.
   (c) **P_k computes its output.**
   P_k computes
   $$L_k = \sum_{i \in [n]} S_{\gamma+k,i}.$$
   (d) If P_k broadcasts $corrupted_j$ for some j during step 1(b) of this stage, then $P_i, i \in [n] \setminus \{k\}$, aborts.
2. $P_i, i \in [n]$, outputs L_i.

Fig. 7: The output generation stage.

The value of the output wire w_{z_δ} is

$$\begin{aligned} H_{z_\delta} &= \sum_{i \in [n]} S_{z_\delta,i} \\ &= \sum_{i \in [n]} (S_{u_\delta,i} + S_{v_\delta,i}) \\ &= \sum_{i \in [n]} S_{u_\delta,i} + \sum_{i \in [n]} S_{v_\delta,i} \\ &= H_{u_\delta} + H_{v_\delta}, \end{aligned}$$

as required.

Case 2: g_δ is a multiplication-by-constant gate.
In this case, each party $P_i, i \in [n]$, sets

$$S_{z_\delta,i} = q_\delta \cdot S_{u_\delta,i}.$$

The value of the output wire w_{z_δ} is

$$\begin{aligned} H_{z_\delta} &= \sum_{i \in [n]} S_{z_\delta,i} \\ &= \sum_{i \in [n]} q_\delta \cdot S_{u_\delta,i} \\ &= q_\delta \cdot \sum_{i \in [n]} S_{u_\delta,i} \\ &= q_\delta \cdot H_{u_\delta}, \end{aligned}$$

as required.

Case 3: g_δ is a multiplication gate.

At step 4(j), P_i computes the following for each $j \in [n] \setminus \{i\}$.

$$
\begin{aligned}
W_{j,i} &= D_{SK_i}\left(V_{j,i}\right) \\
&= D_{SK_i}\left(\sum_{k,\ell\in\{1,2\}^2} K_{j,i,k,\ell}\right) \\
&= D_{SK_i}\left(\sum_{k,\ell\in\{1,2\}^2} A_{j,k} \times_h Y_{i,\ell} +_h E_{PK_i}(H_{j,i,k,\ell}, u_{j,i,k,\ell})\right) \\
&= D_{SK_i}\left(\sum_{k,\ell\in\{1,2\}^2} A_{j,k} \times_h E_{PK_i}(B_{i,\ell}, r_{i,\ell}) +_h E_{PK_i}(H_{j,i,k,\ell}, u_{j,i,k,\ell})\right) \\
&= D_{SK_i}\left(\sum_{k,\ell\in\{1,2\}^2} E_{PK_i}((A_{j,k}\cdot B_{i,\ell}), r1_{j,i,k,\ell})) +_h E_{PK_i}(H_{j,i,k,\ell}, u_{j,i,k,\ell})\right) \\
&\quad \text{[by the homomorphic property of the encryption scheme]} \\
&= D_{SK_i}\left(\sum_{k,\ell\in\{1,2\}^2} E_{PK_i}((A_{j,k}\cdot B_{i,\ell} + H_{j,i,k,\ell}), r2_{j,i,k,\ell})\right) \\
&\quad \text{[by the homomorphic property of the encryption scheme]} \\
&= D_{SK_i}\left(E_{PK_i}\left(\left(\sum_{k,\ell\in\{1,2\}^2} (A_{j,k}\cdot B_{i,\ell} + H_{j,i,k,\ell})\right), r3_{j,i}\right)\right) \\
&\quad \text{[by the homomorphic property of the encryption scheme]} \\
&= \sum_{k,\ell\in\{1,2\}^2} (A_{j,k}\cdot B_{i,\ell} + H_{j,i,k,\ell}) \\
&\quad \text{[by the correctness of the encryption scheme]} \\
&= \sum_{k,\ell\in\{1,2\}^2} A_{j,k}\cdot B_{i,\ell} + \sum_{k,\ell\in\{1,2\}^2} H_{j,i,k,\ell} \\
&= A_j B_i + C_{j,i}.
\end{aligned}
$$

Then, P_i computes its share

$$
\begin{aligned}
S_{z_\delta,i} &= A_i B_i + \sum_{j\in[n]\setminus\{i\}} W_{j,i} - \sum_{j\in[n]\setminus\{i\}} C_{i,j} \\
&= A_i B_i + \sum_{j\in[n]\setminus\{i\}} (A_j B_i + C_{j,i}) - \sum_{j\in[n]\setminus\{i\}} C_{i,j} \\
&= \left(\sum_{j\in[n]} A_j\right)\cdot B_i + \sum_{j\in[n]\setminus\{i\}} C_{j,i} - \sum_{j\in[n]\setminus\{i\}} C_{i,j} \\
&= \left(\sum_{j\in[n]} S_{u_\delta,i}\right)\cdot B_i + \sum_{j\in[n]\setminus\{i\}} C_{j,i} - \sum_{j\in[n]\setminus\{i\}} C_{i,j} \\
&= H_{u_\delta}\cdot B_i + \sum_{j\in[n]\setminus\{i\}} C_{j,i} - \sum_{j\in[n]\setminus\{i\}} C_{i,j}
\end{aligned}
$$

Then it follows that

$$\begin{aligned}
H_{z_\delta} &= \sum_{i\in[n]} S_{z_\delta,i} \\
&= \sum_{i\in[n]} \left(H_{u_\delta} B_i + \sum_{j\in[n]\setminus\{i\}} C_{j,i} - \sum_{j\in[n]\setminus\{i\}} C_{i,j} \right) \\
&= H_{u_\delta} \cdot \sum_{i\in[n]} B_i + \sum_{i\in[n]} \sum_{j\in[n]\setminus\{i\}} C_{j,i} - \sum_{i\in[n]} \sum_{j\in[n]\setminus\{i\}} C_{i,j} \\
&= H_{u_\delta} \left(\sum_{j\in[n]} S_{v_\delta,i} \right) + \sum_{i\in[n]} \sum_{j\in[n]} C_{j,i} - \sum_{i\in[n]} C_{i,i} - \sum_{i\in[n]} \sum_{j\in[n]} C_{i,j} + \sum_{i\in[n]} C_{i,i} \\
&= H_{u_\delta} H_{v_\delta} + \sum_{i\in[n]} \sum_{j\in[n]} C_{i,j} - \sum_{i\in[n]} \sum_{j\in[n]} C_{i,j} \\
&= H_{u_\delta} H_{v_\delta},
\end{aligned}$$

as required.

In this way, we prove that the shares of each wire sum up to the correct value of the wire up to all wires including the output wires $w_{\gamma+1}, \ldots, w_{\gamma+n}$.

In the output generation stage, for each $k \in [n]$, the following holds:
if all parties send their correct shares and randomness for the output wire $w_{\gamma+k}$ to P_k, then P_k can compute its output L_k correctly.

This completes the correctness proof of protocol *Circuit*.

6 Security of protocol *Circuit*

In [?] it is proved that a protocol that securely computes some functionality f_1 with abort in the presence of active adversaries, securely computes f_1 in the presence of covert adversaries with ϵ-deterrent for every ϵ such that $0 \le \epsilon \le 1$. Then we can consider the secure protocols for the active adversary models for functionality f_{CF}, f_{CC} and f_{OC} to be secure protocols with deterrent 1 for the corresponding functionalities in the presence of a covert adversary.

We will prove the security of protocol *Circuit* in the $((f_{CF}, 1), (f_{CC}, 1), (f_{OC}, 1))$-hybrid world.

We have the following Theorem on the security of protocol *Circuit*.

Theorem 2. *Assuming the existence of lossy additive homomorphic public key encryption schemes with efficient Opener algorithm and secure coin tossing protocols in the presence of a malicious adversary, protocol Circuit securely computes functionality f with deterrent $\frac{1}{2}$ in the $((f_{CF}, 1), (f_{CC}, 1), (f_{OC}, 1))$-hybrid world in the presence of a static covert adversary that can corrupt up to $(n-1)$ parties.*

Let M denote the number of multiplication gates in the arithmetic circuit representing f. Protocol Circuit runs in $O(M \log n)$ rounds and needs $O(Mn^2 s)$ communication among the parties where s is the security parameter.

Let $\mathcal{A}_h$ be a static covert adversary that interacts with parties running protocol *Circuit* in the $((f_{CF}, 1), (f_{CC}, 1), (f_{OC}, 1))$-hybrid world. We will construct an ideal-world adversary or simulator $\mathcal{S}$ that runs in time polynomial of the running time of $\mathcal{A}_h$ and such that the following holds for each $I \subset [n]$:

$$\begin{aligned}
&\left\{ IDEAL^{\epsilon}_{f,\mathcal{S}(z),I}(\overline{x}, s) \right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}} \\
\overset{C}{\equiv} &\left\{ HYBRID^{(f_{CF},1),(f_{CC},1),(f_{OC},1)}_{Circuit,\mathcal{A}_h(z),I}(\overline{x}, s) \right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}} \qquad (1)
\end{aligned}$$

where $\overline{x}$ is a balanced vector.

Here we describe the main intuition behind the security of protocol *Circuit*.

In the *key generation stage*, for each honest party, the simulator generates one lossy pair and one injective pair and rewinds until the injective pair is opened for verification. By the “key indistinguishability” property of the lossy encryption scheme, public keys of a lossy pair and public key of an injective pair are computationally indistinguishable. By the “key detection” property of the lossy encryption scheme, a corrupted party attempting to cheat by using lossy keys gets caught with probability at least $\frac{1}{2}$.

In the *input sharing stage*, the simulator uses zero as the inputs of the honest parties. By the semantic security of the lossy encryption scheme, the ciphertexts are indistinguishable in the hybrid world and the ideal world.

During the computation of a multiplication gate, exactly half of the subshares are opened for verification – the distribution of half of the subshares are identical in two worlds. Since exactly half of the calculations are checked for verification, a party attempting cheat in any calculation during the multiplication gets caught with probability at least $\frac{1}{2}$. In each stage of protocol *Circuit*, the simulator checks the responses of the adversary for both values of the challenge, by rewinding the challenge generation step. If the adversary cheats for both values of the challenge, then the simulator simulates catching the cheating party. If the adversary cheats for exactly one value of the challenge and that value is selected for verification, then the simulator simulates catching the cheating party. If the adversary cheats for exactly one value of the challenge and that value is not selected for verification, then the simulation continues to the next stage. If the adversary does not cheat for both values of the challenge, then simulator proceeds to the next stage. In case of *input sharing stage* where the adversary does not cheat for both values of the challenge, the simulator extracts the replaced input of the corrupted parties from the responses of the adversary.

At the *output generation stage*, the simulator sends the inputs of the corrupted parties to the trusted party and receives the actual outputs of the corrupted parties. Then the simulator selects a set of shares for the honest parties satisfying these outputs and the shares of the corrupted parties for the output wires carrying outputs of the corrupted parties. For each such wire and each honest party, the simulator uses the (polynomial time) *Opener* algorithm to compute fake randomness such that encrypting the share of that honest party for that wire using the fake randomness results to the ciphertext held by the adversary for that party for that wire. In any stage of protocol, a party attempting cheat gets caught with probability at least $\frac{1}{2}$.

6.1 A simulator for protocol *Circuit*.

The simulator $\mathcal{S}$ gets the following as its inputs.

1. The set I of corrupted parties,
2. the auxiliary input z,
3. the set of inputs $\{x_i\}_{i\in I}$ of the corrupted parties, and
4. the set of outputs $\{y_i\}_{i\in I}$ of the corrupted parties.

$\mathcal{S}$ invokes $\mathcal{A}_h$ on input

$$\left\{s, I, \{x_i\}_{i\in I}, \{y_i\}_{i\in I}, z\right\}.$$

If the input x_i of P_i is $abort_i$, then $\mathcal{S}$ sends $abort_i$ to the trusted party and halts.

If the input x_i of P_i is $corrupted_i$, then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.

If the input x_i of P_i is $cheat_i$, then $\mathcal{S}$ sends $cheat_i$ to the trusted party. If the trusted party replies with $corrupted_i$ to $\mathcal{S}$ and all the honest parties, then $\mathcal{S}$ halts. Observe that the trusted party replies with $corrupted_i$ with probability $\epsilon = \frac{1}{2}$ according to the definition. If the trusted party replies with *undetected* and the set $\{x_i\}_{i\in[n]:P_i \text{ is honest}}$ of inputs of the honest parties for f to $\mathcal{S}$, then $\mathcal{S}$ sends *undetected* and the set $\{x_i\}_{i\in[n]:P_i \text{ is honest}}$ to $\mathcal{A}_h$. According to the definition, the trusted party replies with $corrupted_i$ and the inputs of the honest parties with probability $1 - \epsilon = \frac{1}{2}$. Then $\mathcal{S}$ receives a vector $\{y'_i\}_{i \in [n]: P_i \text{ is honest}}$ of outputs for the honest parties of the adversary's choice from $\mathcal{A}_h$. $\mathcal{S}$ sends this vector $\{y'_i\}_{i \in [n]: P_i \text{ is honest}}$ to the trusted party and halts.

If no input x_i is $abort_i, corrupted_i$ or $cheat_i$, then $\mathcal{S}$ proceeds as follows. In some steps, $\mathcal{S}$ is supposed to get some message for each corrupted party P_i, from $\mathcal{A}_h$. In those steps, if $\mathcal{S}$ does not receive any message from $\mathcal{A}_h$ on behalf of a corrupted party P_i, then $\mathcal{S}$ sends $abort_i$ to the trusted party and halts.

We design the simulator $\mathcal{S}$ in such a way that the view of any party except a single party in the simulation is guaranteed to be computationally indistinguishable to its view in the real world. The identity of the single inconsistent party is generated uniformly at random at the start of the simulation. We call this step the SIP (single inconsistent party) Fixation step.

SIP Fixation step: The simulator $\mathcal{S}$ generates s, the identity of the SIP, uniformly at random from $[n]$. During the simulation, if the adversary generates a request to corrupt P_s, then $\mathcal{S}$ rewinds to this step and generates a new random index s' uniformly at random from $[n]$ and proceeds again. Since the adversary is t-limited where $t < (n-1)/2$, it holds that the probability of a randomly selected party P_s being corrupted is at most $\frac{1}{2}$. So the expected number of rewinds of the simulator is at most two and the simulation can be performed in expected polynomial time. To bound the running time of the simulator $\mathcal{S}$ to strictly polynomial time, we can continue running upto κ^{ℓ_1} steps where ℓ_1 is a predetermined constant. If $\mathcal{S}$ does not halt within κ^{ℓ_1} steps, then $\mathcal{S}$ fails. The probability of faliure of $\mathcal{S}$ is negligible.

In the simulation, after each step we describe a special step denoted as the modification upon corruption – this step describes how the simulator modifies some of its already calculated values if an honest party gets corrupted after completing the corresponding step. We separately describe with this tagging to clarify the description of the simulation. In most cases, the modification steps performed at an earlier step is also performed as part of the modification steps in a later step. For easier referencing, we tag the modification steps after each step.

Simulating the CRS generation stage

1. For each corrupted party $P_i, i\mathcal{S}$ receives the input uc_i of P_i for this stage from $\mathcal{A}_h$. If $uc_i = abort_i$, then $\mathcal{S}$ sends $abort_i$ to the trusted party and halts.
2. $\mathcal{S}$ generates a string σ_κ of length $p_1(\kappa)$ along with its trapdoor so that $\mathcal{S}$ can control the outcome of the commitment protocol $CommittedCoinFlip_\sigma$ in later stages of protocol $Circuit$.
3. For each corrupted party $P_i, \mathcal{S}$ sends σ_κ as the output of P_i for this stage to $\mathcal{A}_h$.

Simulating the key generation stage

1. $\mathcal{S}$ generates a key pair
$$(PK_p, SK_p) = ((g_1, N), \lambda) \leftarrow KeyGenPaillier(1^\kappa)$$
where $KeyGenPaillier$ denotes the key generation algorithm of (non-threshold) Paillier encryption scheme.
2. $\mathcal{S}$ selects $r_q \xleftarrow{\$} \mathbb{Z}_N^*$.
3. $\mathcal{S}$ sets $b = 0$ (for lossy mode, $b = 0$).
4. $\mathcal{S}$ computes
$$\begin{aligned} Q &= g_1^b r_q^N \bmod N^2 \\ &= g_1^0 r_q^N \bmod N^2 \\ &= r_q^N \bmod N^2. \end{aligned}$$
5. $\mathcal{S}$ invokes the simulator $\mathcal{S}_{KG}$ of key generation of lossy threshold Paillier encryption scheme on input (g_1, N, Q, s), with one minor modification in the code of $\mathcal{S}_{KG}$. The modification is that $\mathcal{S}_{KG}$ also returns the trapdoor ct_j of the commitment key ck_j of each party P_j, to $\mathcal{S}$.

6. $\mathcal{S}$ receives

$$\big((g,N,\theta,Q),\{s_j\}_{j\in[n]},v,\{v_i\}_{i\in[n]},\{ck_i\}_{i\in[n]},\{ct_i\}_{i\in[n]},\{f_i(j)\}_{i,j\in[n]},\{w_{i,j}\}_{i,j\in[n]},\{r_{i,j}\}_{i,j\in[n]}\big)$$

from $\mathcal{S}_{KG}$.
7. For each corrupted party P_j, $\mathcal{S}$ sends

$$key_j=\big((g,N,\theta,Q),s_j,v,\{v_i\}_{i\in[n]},\{ck_i\}_{i\in[n]},\{f_i(j)\}_{i\in[n]},\{w_{i,\ell}\}_{i,\ell\in[n]},\{r_{i,j}\}_{i\in[n]}\big)$$

as the output of P_j for this stage, to $\mathcal{A}$.
8. $\mathcal{S}$ computes the secret key,

$$SK=\beta\lambda=\sum_{j\in[n]}s_j.$$

9. $\mathcal{S}$ sets PK to (g,N,θ,Q).

Simulating the input sharing stage

1. Simulating the challenge generation step of input sharing stage.
   (a) If the input of a party P_i is $abort_i$ for this step, then $\mathcal{S}$ sends $abort_i$ to the trusted party and halts.
   (b) $\mathcal{S}$ runs the simulator $S_{CommittedCoinFlipPublic_\sigma}$ for the $CommittedCoinFlipPublic_\sigma$ protocol, using σ_κ as the common input and 0 as the input of each honest parties.
   If a party P_i aborts during the execution of $CommittedCoinFlipPublic_\sigma$, then $\mathcal{S}$ sends $abort_i$ to the trusted party and halts. If $S_{CommittedCoinFlipPublic_\sigma}$ catches a cheating party P_i, then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.
   If no party aborts or gets caught, then $\mathcal{S}$ extracts the committed values of the corrupted parties from $S_{CommittedCoinFlipPublic_\sigma}$.
2. Simulating the share broadcasting step.
   (a) For each honest party P_i, $\mathcal{S}$ performs the following steps.
      i. $\mathcal{S}$ randomly selects two sets $\{B_{1,i,j}\}_{j\in[n]}$ and $\{B_{2,i,j}\}_{j\in[n]}$satisfying

$$\sum_{j\in[n]}B_{1,i,j}=\sum_{j\in[n]}B_{2,i,j}=0.$$

      ii. $\mathcal{S}$ randomly selects two sets of strings $\{b_{1,i,j}\}_{j\in[n]}$ and $\{b_{2,i,j}\}_{j\in[n]}$.
      iii. For each $\ell\in\{1,2\}$ and each $j\in[n]$, $\mathcal{S}$ sends

$$Y_{\ell,i,j}=E_{PK}\left(B_{\ell,i,j},b_{\ell,i,j}\right)$$

      to $\mathcal{A}_h$.
      Modification upon corruption: If P_i gets corrupted after this step, then $\mathcal{S}$ performs the following steps.
      A. $\mathcal{S}$ corrupts P_i in the ideal world and receives its input x_i from $\mathcal{Z}$.
      B. $\mathcal{S}$ computes

$$B_{1,i,i}=x_i-\sum_{j\in[n]\setminus\{i\}}B_{1,i,j},$$

      and

$$B_{2,i,i}=x_i-\sum_{j\in[n]\setminus\{i\}}B_{2,i,j}.$$

      C. $\mathcal{S}$ computes

$$b_{1,i,i}=Opener(PK,SK,B_{1,i,i},Y_{1,i,i})$$

      and

$$b_{2,i,i}=Opener(PK,SK,B_{2,i,i},Y_{2,i,i}).$$

   (b) For each corrupted party P_i, each $\ell\in\{1,2\}$ and each $j\in[n]$, $\mathcal{S}$ receives $Y_{\ell,i,j}$ from $\mathcal{A}_h$.
3. Simulating the share sending step.

(a) For each honest party P_i and each corrupted party $P_j, \mathcal{S}$ sends the set $\{B_{1,i,j}, B_{2,i,j}, b_{1,i,j}, b_{2,i,j}\}$ to $\mathcal{A}_h$.

(b) For each corrupted party P_i and each honest party $P_j, \mathcal{S}$ receives the set $\{B_{1,i,j}, B_{2,i,j}, b_{1,i,j}, b_{2,i,j}\}$ from $\mathcal{A}_h$.

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ performs the same steps listed as the modifications upon corruption at step 2(a)(iii).

4. Simulating the challenge opening step of input sharing stage.
$\mathcal{S}$ runs the simulator $S_{OpenCom_\sigma}$ of $OpenCom_\sigma$, opening the commitments of the honest parties to zero. If a party P_i aborts during the execution of $OpenCom_\sigma$, then $\mathcal{S}$ sends $abort_i$ to the trusted party and halts. If $S_{OpenCom_\sigma}$ catches a cheating party P_i, then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ performs the same steps listed as the modifications upon corruption at step 2(a)(iii).

5. Simulating the challenge response step of input sharing stage.
(a) For each honest party $P_i, \mathcal{S}$ sends the sets $\{B_{m_{in},i,j}\}_{j\in[n]\setminus\{i\}}$ and $\{b_{m_{in},i,j}\}_{j\in[n]\setminus\{i\}}$ to $\mathcal{A}_h$.
(b) For each corrupted party P_i, $\mathcal{S}$ receives the sets $\{B_{m_{in},i,j}\}_{j\in[n]\setminus\{i\}}$ and $\{b_{m_{in},i,j}\}_{j\in[n]\setminus\{i\}}$ from $\mathcal{A}_h$.
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ performs the same steps listed as the modifications upon corruption at step 2(a)(iii).

6. Simulating the verification step of input sharing stage.
$\mathcal{S}$ verifies the responses of the corrupted parties.
Then $\mathcal{S}$ rewinds back to step 1 and instructs the simulator $S_{CommittedCoinFlipPublic_\sigma}$ so that the challenge takes on the value $(3 - m_{in})$(which is the other possible value of the challenge). For this new challenge, $\mathcal{S}$ performs steps $2 - 5$ and verifies the responses of the corrupted parties.
$\mathcal{S}$ then performs the following actions depending on the responses, as described below.
Case 1: No violation occurs during any of the simulations.
For each corrupted party $P_i, \mathcal{S}$ performs the following steps.
(a) $\mathcal{S}$ computes
$$B_{1,i,i} = D_{SK}(Y_{1,i,i}) = D_{\beta\lambda}(Y_{1,i,i}) = \frac{L\left((Y_{1,i,i})^{\beta\lambda}\right)}{\theta} \bmod N$$
and
$$B_{2,i,i} = D_{SK}(Y_{2,i,i}) = D_{\beta\lambda}(Y_{2,i,i}) = \frac{L\left((Y_{2,i,i})^{\beta\lambda}\right)}{\theta} \bmod N.$$
(b) $\mathcal{S}$ computes
$$x'_i = \sum_{j\in[n]} B_{3-m_{in},i,j}.$$
Case 2: There is some violation in exactly one simulation.
Let P_i be some party for which $\mathcal{A}_h$ provided invalid response in the simulation corresponding to the challenge m_{ci} where $m_{ci} \in \{1,2\}$.
If $m_{ci} = m_{in}$, then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.
If $m_{ci} \neq m_{in}$, then $\mathcal{S}$ first computes x' as in case 1, then proceeds to the next step (step 7).
Case 3: There is some violation in both of the simulations.
Let P_i be some party for which $\mathcal{A}_h$ provided invalid responses. Then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.

7. Simulating the share fixation step of input sharing stage.
For each $k \in [n]$ and each $i \in [n], \mathcal{S}$ performs the following steps.
(a) $\mathcal{S}$ sets $S_{k,i}$ to $B_{3-m_{in},k,i}$.
(b) $\mathcal{S}$ sets $rs_{k,i}$ to $b_{3-m_{in},k,i}$.
(c) For each $j \in [n] \setminus \{i\}, \mathcal{S}$ sets $ES_{k,i,j}$ to $Y_{3-m_{in},k,i}$.
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 2(a)(iii), then sets $S_{i,i}$ and $rs_{i,i}$ to the modified value of $B_{3-m_{in},i,i}$ and $b_{3-m_{in},i,i}$, resepectively.

Simulating the computation stage.

Before describing the simulation of this stage, we first describe the following sub-routine called patching. In the computation stgae, parties evaluate the gates in the order $(g_1, \ldots, g_\theta)$. If an honest party P_i gets corrupted at any step during the evaluation of gate g_δ, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of the input sharing stage. Then, $\mathcal{S}$ have to patch the gates evaluated so far to the actual input of P_i. For convenience of description, we list the actions performed by $\mathcal{S}$ for patching the states for already evaluated gates $(g_1, \ldots, g_{\delta-1})$ in this subroutine.

Patching:

For each $\mu \in \{1, \ldots, \delta - 1\}, \mathcal{S}$ performs the following steps.

Case 1: g_μ is an addition gate.

1. $\mathcal{S}$ sets
$$S_{z_\mu,i} = S_{u_\mu,i} + S_{v_\mu,i}.$$
2. $\mathcal{S}$ sets
$$rs_{z_\mu,i} = \mathit{Opener}(PK, SK, S_{z_\mu,i}, ES_{z_\mu,\ell,i})$$
where $\ell \in [n] \setminus \{i\}$.

Case 2: g_μ is a multiplication-by-constant gate.

1. $\mathcal{S}$ sets
$$S_{z_\mu,i} = q_\mu \times S_{u_\mu,i}.$$
2. $\mathcal{S}$ sets
$$rs_{z_\mu,i} = \mathit{Opener}(PK, SK, S_{z_\mu,i}, ES_{z_\mu,\ell,i})$$
where $\ell \in [n] \setminus \{i\}$.

Case 3: g_μ is a multiplication gate.

1. $\mathcal{S}$ sets
$$A^{(\mu)}_{i,3-m^{(\mu)}} = S_{u_\mu,i} - A^{(\mu)}_{i,m^{(\mu)}},$$
$$a^{(\mu)}_{i,3-m^{(\mu)}} = \mathit{Opener}\left(PK, SK, A^{(\mu)}_{i,3-m^{(\mu)}}, X^{(\mu)}_{i,3-m^{(\mu)}}\right),$$
$$B^{(\mu)}_{i,3-m^{(\mu)}} = S_{v_\mu,i} - B^{(\mu)}_{i,m^{(\mu)}},$$
$$b^{(\mu)}_{i,3-m^{(\mu)}} = \mathit{Opener}\left(PK, SK, B^{(\mu)}_{i,3-m^{(\mu)}}, Y^{(\mu)}_{i,3-m^{(\mu)}}\right).$$
2. For each $\ell \in \{1, 2\}, \mathcal{S}$ sets
$$vv^{(\mu)}_{i,3-m^{(\mu)},\ell} = \mathit{Opener}\left(PK, SK, 0, \left(L^{(\mu)}_{i,3-m^{(\mu)},\ell} -_h A^{(\mu)}_{i,3-m^{(\mu)}} \times_h Y^{(\mu)}_{i,\ell}\right)\right).$$
3. For each $j \in [n] \setminus \{i\}$ and each $\ell \in \{1, 2\}, \mathcal{S}$ sets
$$hh^{(\mu)}_{i,j,3-m^{(\mu)},\ell} = \mathit{Opener}\left(PK, SK, H^{(\mu)}_{i,j,3-m^{(\mu)},\ell}, \left(K^{(\mu)}_{i,j,3-m^{(\mu)},\ell} -_h A^{(\mu)}_{i,3-m^{(\mu)}} \times_h Y^{(\mu)}_{j,\ell}\right)\right).$$
4. $\mathcal{S}$ computes
$$val_\mu = \left(\sum_{j\in[n]} S_{u_\mu,j}\right) \cdot \left(\sum_{j\in[n]} S_{v_\mu,j}\right).$$
5. $\mathcal{S}$ computes
$$S_{z_\mu,i} = val_\mu - \sum_{j\in[n]\setminus\{i\}} S_{z_\mu,j}.$$

6. $\mathcal{S}$ computes
$$rs_{z_\mu,i} = \mathit{Opener}\left(PK, SK, S_{z_\mu,j}, ES_{z_\mu,\ell,i}\right)$$
where $\ell \in [n] \setminus \{i\}$.

Now we describe the action of $\mathcal{S}$ for simulating the corruption stage.

For each $\delta \in \{1, \ldots, \theta\}, \mathcal{S}$ performs the following actions, depending on the type of gate g_δ.

Case 1: g_δ is an addition gate.
For each $i \in [n], \mathcal{S}$ performs the following actions.

1. $\mathcal{S}$ sets
$$S_{z_\delta,i} = S_{u_\delta,i} + S_{v_\delta,i}.$$
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*. Then $\mathcal{S}$ sets
$$S_{z_\delta,i} = S_{u_\delta,i} + S_{v_\delta,i}.$$
2. $\mathcal{S}$ computes $rs_{z_\delta,i}$ such that the following equality holds.
$$E_{PK}\left(S_{u_\delta,i}, rs_{u_\delta,i}\right) +_h E_{PK}\left(S_{v_\delta,i}, rs_{v_\delta,i}\right) = E_{PK}\left(\left(S_{u_\delta,i} + S_{v_\delta,i}\right), rs_{z_\delta,i}\right).$$
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*. Then $\mathcal{S}$ sets
$$S_{z_\delta,i} = S_{u_\delta,i} + S_{v_\delta,i},$$
and $\mathcal{S}$ computes $rs_{z_\delta,i}$ such that the following equality holds.
$$E_{PK}\left(S_{u_\delta,i}, rs_{u_\delta,i}\right) +_h E_{PK}\left(S_{v_\delta,i}, rs_{v_\delta,i}\right) = E_{PK}\left(\left(S_{u_\delta,i} + S_{v_\delta,i}\right), rs_{z_\delta,i}\right).$$
3. For each $j \in [n] \setminus \{i\}$, $\mathcal{S}$ sets
$$ES_{u_\delta,i,j} = ES_{u_\delta,i,j} +_h ES_{v_\delta,i,j}.$$
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*. Then $\mathcal{S}$ sets
$$S_{z_\delta,i} = S_{u_\delta,i} + S_{v_\delta,i},$$
and $\mathcal{S}$ computes $rs_{z_\delta,i}$ such that the following equality holds.
$$E_{PK}\left(S_{u_\delta,i}, rs_{u_\delta,i}\right) +_h E_{PK}\left(S_{v_\delta,i}, rs_{v_\delta,i}\right) = E_{PK}\left(\left(S_{u_\delta,i} + S_{v_\delta,i}\right), rs_{z_\delta,i}\right).$$

Case 2: g_δ is a multiplication-by-constant gate.
Let $q_\delta \in \mathbb{F}$ be the constant with which the multiplication needs to be done.
For each $i \in [n], \mathcal{S}$ performs the following actions.

1. $\mathcal{S}$ sets
$$S_{z_\delta,i} = q_\delta \cdot S_{u_\delta,i}.$$
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*. Then $\mathcal{S}$ sets
$$S_{z_\delta,i} = q_\delta \cdot S_{u_\delta,i}.$$

2. $\mathcal{S}$ computes $rs_{z_\delta,i}$ such that the following equality holds.
$$q_\delta \times_h E_{PK}\left(S_{u_\delta,i}, rs_{u_\delta,i}\right) = E_{PK}\left(\left(q_\delta \cdot S_{u_\delta,i}\right), rs_{z_\delta,i}\right).$$
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*. Then $\mathcal{S}$ sets
$$S_{z_\delta,i} = q_\delta \cdot S_{u_\delta,i},$$
and $\mathcal{S}$ computes $rs_{z_\delta,i}$ such that the following equality holds.
$$q_\delta \times_h E_{PK}\left(S_{u_\delta,i}, rs_{u_\delta,i}\right) = E_{PK}\left(\left(q_\delta \cdot S_{u_\delta,i}\right), rs_{z_\delta,i}\right).$$
3. For each $j \in [n] \setminus \{i\}$, $\mathcal{S}$ sets
$$ES_{u_\delta,i,j} = q_\delta \cdot ES_{u_\delta,i,j}.$$
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*. Then $\mathcal{S}$ sets
$$S_{z_\delta,i} = q_\delta \cdot S_{u_\delta,i},$$
and $\mathcal{S}$ computes $rs_{z_\delta,i}$ such that the following equality holds.
$$q_\delta \times_h E_{PK}\left(S_{u_\delta,i}, rs_{u_\delta,i}\right) = E_{PK}\left(\left(q_\delta \cdot S_{u_\delta,i}\right), rs_{z_\delta,i}\right).$$

Case 3: g_δ is a multiplication gate.

1. (a) i. If the input of a party P_i is $abort_i$ for this step, then $\mathcal{S}$ sends $abort_i$ to the trusted party and halts.
   ii. S runs the simulator $S_{CommittedCoinFlipPublic_\sigma}$ for the $CommittedCoinFlipPublic_\sigma$ protocol, using σ_κ as the common input and 0 as the input of each honest parties.
   If a party P_i aborts during the execution of $CommittedCoinFlipPublic_\sigma$, then $\mathcal{S}$ sends $abort_i$ to the trusted party and halts. If $S_{CommittedCoinFlipPublic_\sigma}$ catches a cheating party P_i, then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.
   If no party aborts or gets caught, then $\mathcal{S}$ extracts the committed values of the corrupted parties from $S_{CommittedCoinFlipPublic_\sigma}$.
   Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*.
   (b) For each honest party $P_i, \mathcal{S}$ randomly selects two sets of shares $\{Q_{1,i,j}\}_{j\in[n]\setminus\{i\}}, \{Q_{2,i,j}\}_{j\in[n]\setminus\{i\}}$ and two sets of strings $\{rq_{1,i,j}\}_{j\in[n]\setminus\{i\}}, \{rq_{2,i,j}\}_{j\in[n]\setminus\{i\}}$.
   Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*.
   (c) i. For each honest party P_i, each $\ell \in \{1,2\}$ and each $j \in [n] \setminus \{i\}, \mathcal{S}$ sends
   $$Y_{\ell,i,j} = E_{PK}(Q_{\ell,i,j}, rq_{\ell,i,j})$$
   to $\mathcal{A}_h$.
   Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*.
   ii. For each corrupted party P_i, each $\ell \in \{1,2\}$ and each $j \in [n] \setminus \{i\}, \mathcal{S}$ receives $Y_{\ell,i,j}$ from $\mathcal{A}_h$.

(d) $\mathcal{S}$ runs the simulator $S_{OpenCom_\sigma}$ of $OpenCom_\sigma$, opening the commitments of the honest parties to zero. If a party P_i aborts during the execution of $OpenCom_\sigma$, then $\mathcal{S}$ sends $abort_i$ to the trusted party and halts. If $S_{OpenCom_\sigma}$ catches a cheating party P_i, then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*.

(e) i. For each honest party P_i and each $j \in [n] \setminus \{i\}$, $\mathcal{S}$ sends $Q_{m_r,i,j}$ and $rq_{m_r,i,j}$ to $\mathcal{A}_h$.

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*.

ii. For each corrupted party P_i and each $j \in [n] \setminus \{i\}$, $\mathcal{S}$ receives $Q_{m_r,i,j}$ and $rq_{m_r,i,j}$ from $\mathcal{A}_h$.

(f) $\mathcal{S}$ verifies the responses of the corrupted parties.

Then $\mathcal{S}$ rewinds back to step 1(a) and instructs the simulator $S_{CommittedCoinFlipPublic_\sigma}$ so that the challenge takes on the value $(3 - m_r)$(which is the other possible value of the challenge). For this new challenge, $\mathcal{S}$ performs steps $1(a) - 1(e)$ and verifies the responses of the corrupted parties.

$\mathcal{S}$ then performs the following actions depending on the responses, as described below.

Case 1: No violation occurs during any of the simulations.
$\mathcal{S}$ proceeds to the next step (step 1(g)).

Case 2: There is some violation in exactly one simulation.
Let P_i be some party for which $\mathcal{A}_h$ provided invalid response in the simulation corresponding to the challenge m_{cr} where $m_{cr} \in \{1, 2\}$.
If $m_{cr} = m_r$, then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.
If $m_{cr} \neq m_r$, then $\mathcal{S}$ proceeds to the next step (step 1(g)).

Case 3: There is some violation in both of the simulations.
Let P_i be some party for which $\mathcal{A}_h$ provided invalid responses. Then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*.

(g) For each $i \in [n]$ and each $j \in [n] \setminus \{i\}, \mathcal{S}$ performs the following steps.

i. $\mathcal{S}$ sets $C^{(\delta)}_{i,j}$ to $Q_{3-m_r,i,j}$ and $rc^{(\delta)}_{i,j}$ to $rq_{3-m_r,i,j}$.

ii. For each $k \in [n] \setminus \{j\}, \mathcal{S}$ sets $EC^{(\delta)}_{i,j,k}$ to $Y_{3-m_r,j,k}$.

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*.

2. $\mathcal{S}$ runs the simulator $S_{CommittedCoinFlipPublic_\sigma}$ for the $CommittedCoinFlipPublic_\sigma$ protocol, using 0 as the inputs of the honest parties.

If a party P_i aborts during the execution of $CommittedCoinFlipPublic_\sigma$, then $\mathcal{S}$ sends $abort_i$ to the trusted party and halts. If $S_{CommittedCoinFlipPublic_\sigma}$ catches a cheating party P_i, then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*.

3. (a) i. $\mathcal{S}$ randomly selects $d \in \{1, 2\}$.

ii. For each honest party $P_i, \mathcal{S}$ chooses random field elements $A^{(\delta)}_{i,d}, B^{(\delta)}_{i,d} \in \mathbb{F}$.

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*.

(b) S performs nothing.

(c) For each honest party $P_i, \mathcal{S}$ chooses random strings $a^{(\delta)}_{i,d}, b^{(\delta)}_{i,d}, aa^{(\delta)}_{i,0}$ and $bb^{(\delta)}_{i,0}$.

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*.

(d) i. For each honest party $P_i, \mathcal{S}$ computes and sends the following to $\mathcal{A}_h$.

$$X_{i,d}^{(\delta)} = E_{PK}\left(A_{i,d}^{(\delta)}, a_{i,d}^{(\delta)}\right),$$

$$X_{i,3-d}^{(\delta)} = EA_{\ell,i}^{(\delta)} -_h X_{i,d}^{(\delta)} -_h E_{PK}\left(0, aa_{i,0}^{(\delta)}\right)$$

where $\ell \in [n] \setminus \{i\}$.

$$Y_{i,d}^{(\delta)} = E_{PK}\left(B_{i,d}^{(\delta)}, b_{i,d}^{(\delta)}\right),$$

$$Y_{i,3-d}^{(\delta)} = EB_{\ell,i}^{(\delta)} -_h Y_{i,d}^{(\delta)} -_h E_{PK}\left(0, bb_{i,0}^{(\delta)}\right)$$

where $\ell \in [n] \setminus \{i\}$.

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ first performs the modification upon corruption steps listed in step 7 of input sharing stage. Then $\mathcal{S}$ performs the actions listed in *Patching*.

Then $\mathcal{S}$ sets

$$A_{i,3-d}^{(\delta)} = S_{u_\delta,i} - A_{i,d}^{(\delta)},$$

$$a_{i,3-d}^{(\delta)} = \mathit{Opener}\left(PK, SK, A_{i,3-d}^{(\delta)}, X_{i,3-d}^{(\delta)}\right),$$

$$B_{i,3-d}^{(\delta)} = S_{v_\delta,i} - B_{i,d}^{(\delta)},$$

$$b_{i,3-d}^{(\delta)} = \mathit{Opener}\left(PK, SK, B_{i,3-d}^{(\delta)}, Y_{i,3-d}^{(\delta)}\right).$$

ii. For each corrupted party P_i, $\mathcal{S}$ receives $\left\{X_{i,j}^{(\delta)}, Y_{i,j}^{(\delta)}\right\}_{j\in\{1,2\}}$ from $\mathcal{A}_h$.

(e) i. For each honest party P_i and each $j \in [n] \setminus \{i\}, \mathcal{S}$ performs the following steps.

– For each $k, \ell \in \{1,2\}^2$, $\mathcal{S}$ chooses a random field element $H_{i,j,k,\ell}^{(\delta)}$, a random string $h_{i,j,k,\ell}^{(\delta)}$, then computes and sends the following to $\mathcal{A}_h$.

$$G_{i,j,k,\ell}^{(\delta)} = E_{PK}\left(H_{i,j,k,\ell}^{(\delta)}, h_{i,j,k,\ell}^{(\delta)}\right).$$

– P_i computes a string $cc_{i,j,0}^{(\delta)}$ such that

$$E_{PK}\left(0, cc_{i,j,0}^{(\delta)}\right) = E_{PK}\left(C_{i,j}^{(\delta)}, rc_{i,j}^{(\delta)}\right) -_h G_{i,j,1,1}^{(\delta)} -_h G_{i,j,1,2}^{(\delta)} -_h G_{i,j,2,1}^{(\delta)} -_h G_{i,j,2,2}^{(\delta)}.$$

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 3(d).

ii. For each corrupted party P_i, $\mathcal{S}$ receives $\left\{G_{i,j,k,\ell}^{(\delta)}\right\}_{j\in[n]\setminus\{i\},(k,\ell)\in\{1,2\}^2}$ from $\mathcal{A}_h$.

4. (a) For each honest party $P_i, \mathcal{S}$ sends the strings $aa_{i,0}^{(\delta)}, bb_{i,0}^{(\delta)}$ selected earlier and the set $\left\{cc_{i,j,0}^{(\delta)}\right\}_{j\in[n]\setminus\{i\}}$ computed above, to $\mathcal{A}_h$.

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 3(d).

(b) For each corrupted party $P_i, \mathcal{S}$ receives the strings $aa_{i,0}^{(\delta)}, bb_{i,0}^{(\delta)}$ and the set $\left\{cc_{i,j,0}^{(\delta)}\right\}_{j\in[n]\setminus\{i\}}$ of strings, from $\mathcal{A}_h$.

5. (a) For each honest party P_i and each $(k,\ell) \in \{1,2\}^2, \mathcal{S}$ honestly computes and sends $L_{i,k,\ell}^{(\delta)}$ to $\mathcal{A}_h$.

Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 3(d). Then $\mathcal{S}$ sets

$$vv_{i,3-d,\ell}^{(\delta)} = \mathit{Opener}\left(PK, SK, 0, \left(L_{i,3-d,\ell}^{(\delta)} -_h A_{i,3-d}^{(\delta)} \times_h Y_{i,\ell}^{(\delta)}\right)\right)$$

for each $\ell \in \{1,2\}$.

(b) For each corrupted party P_i and each $(k,\ell) \in \{1,2\}^2, \mathcal{S}$ receives $L^{(\delta)}_{i,k,\ell}$ from $\mathcal{A}_h$.

6. (a) For each honest party P_i, each $j \in [n] \setminus \{i\}$ and each $(k,\ell) \in \{1,2\}^2, \mathcal{S}$ honestly computes and sends $K^{(\delta)}_{i,j,k,\ell}$ to $\mathcal{A}_h$.
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 5. Then $\mathcal{S}$ sets
$$hh^{(\delta)}_{i,j,3-d,\ell} = Opener\left(PK, SK, H^{(\delta)}_{i,j,3-d,\ell}, \left(K^{(\delta)}_{i,j,3-d,\ell} -_h A^{(\delta)}_{i,3-d} \times_h Y^{(\delta)}_{j,\ell}\right)\right)$$
for each $j \in [n] \setminus \{i\}$ and each $\ell \in \{1,2\}$.
(b) For each corrupted party P_i, each $j \in [n] \setminus \{i\}$ and each $(k,\ell) \in \{1,2\}^2, \mathcal{S}$ receives $K^{(\delta)}_{i,j,k,\ell}$ from $\mathcal{A}_h$.

7. $\mathcal{S}$ runs the simulator $S_{OpenCom_\sigma}$ of $OpenCom_\sigma$, opening the commitments of the honest parties to zero. If a party P_i aborts during the execution of $OpenCom_\sigma$, then $\mathcal{S}$ sends $abort_i$ to the trusted party and halts. If $S_{OpenCom_\sigma}$ catches a cheating party P_i, then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 6.

8. If $d = m^{(\delta)}$, then, for each honest party P_i, $\mathcal{S}$ sends
$$\left\{A^{(\delta)}_{i,m^{(\delta)}}, B^{(\delta)}_{i,m^{(\delta)}}, a^{(\delta)}_{i,m^{(\delta)}}, b^{(\delta)}_{i,m^{(\delta)}}, vv^{(\delta)}_{i,m^{(\delta)},1}, vv^{(\delta)}_{i,m^{(\delta)},2}, \left\{H^{(\delta)}_{i,j,m^{(\delta)},\ell}, h^{(\delta)}_{i,j,m^{(\delta)},\ell}, hh^{(\delta)}_{i,j,m^{(\delta)},\ell}\right\}_{j\in[n]\setminus\{i\},\ell\in\{1,2\}}\right\},$$
to $\mathcal{A}_h$.
If $d \neq m^{(\delta)}$, then $\mathcal{S}$ rewinds back to step 3 and performs steps 3-8 again.
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 6.

9. $\mathcal{S}$ verifies the responses of the corrupted parties.
Then $\mathcal{S}$ rewinds back to step 2 and instructs the simulator $S_{CommittedCoinFlipPublic_\sigma}$ so that the outcome of step 2 (the challenge generation step) takes on the value $(3 - m^{(\delta)})$(which is the other possible value of the challenge). For this new challenge $\mathcal{S}$ performs steps $2 - 8$ and verifies the responses of the corrupted parties.
$\mathcal{S}$ then performs the following actions depending on the responses, as described below.
Case 1: No violation occurs during any of the simulations.
$\mathcal{S}$ proceeds to the next step (step 10).
Case 2: There is some violation in exactly one simulation.
Let P_i be some party for which $\mathcal{A}_h$ sent invalid response in the simulation corresponding to the challenge m_{cm} where $m_{cm} \in \{1,2\}$.
If $m_{cm} = m^{(\delta)}$, then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts.
If $m_{cm} \neq m^{(\delta)}$, then $\mathcal{S}$ proceeds to the next step (step 10).
Case 3: There is some violation in both of the simulations.
Let P_i be some party for which $\mathcal{A}_h$ provided invalid response in both simulations. $\mathcal{S}$ sends $corrupted_i$ to the trusted party.
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 6.

10. For each honest party P_i and each $j \in [n] \setminus \{i\}, \mathcal{S}$ computes
$$ES_{z_\delta,i,j} = \sum_{k,\ell} L^{(\delta)}_{j,k,\ell} +_h \sum_{k\in[n]\setminus\{j\}} \sum_{\ell_1,\ell_2\in\{1,2\}^2} K^{(\delta)}_{k,j,\ell_1,\ell_2} -_h \sum_{k\in[n]\setminus\{j\}} EC^{(\delta)}_{i,j,k}.$$
Modification upon corruption: If an honest party P_i gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 6.

11. For each $j \in [n], \mathcal{S}$ performs the following two steps.
– $\mathcal{S}$ computes
$$cip_{\delta,j} = \sum_{k,\ell} L^{(\delta)}_{j,k,\ell} +_h \sum_{k\in[n]\setminus\{j\}} \sum_{\ell_1,\ell_2\in\{1,2\}^2} K^{(\delta)}_{k,j,\ell_1,\ell_2} -_h \sum_{k\in[n]\setminus\{j\}} EC^{(\delta)}_{i,j,k}$$
where $i \in [n] \setminus \{j\}$.

– $\mathcal{S}$ computes

$$msg_{\delta,j} = D_{SK}\left(cip_{\delta,j}\right) = D_{\beta\lambda}\left(cip_{\delta,j}\right) = \frac{L\left(\left(cip_{\delta,j}\right)^{\beta\lambda}\right)}{\theta} \bmod N.$$

For each $j \in [n], \mathcal{S}$ performs the following steps.

(a) If P_j is corrupted, then $\mathcal{S}$ performs the following steps.
   – $\mathcal{S}$ invokes the simulator $\mathcal{S}_D$ on input

$$\left(\{s_i\}_{i\in[n]}, \{v_i\}_{i\in[n]}, \{a_{i,\ell}\}_{i,\ell\in[n]}, \{w_{i,\ell}\}_{i,\ell\in[n]}, \{ck_i\}_{i\in[n]}, g, N, \theta, Q, v, cip_{\delta,j}, msg_{\delta,j}, s, ct_s, \mathcal{A}'\right)$$

   where $\mathcal{A}'$ denote the current state of adversary $\mathcal{A}$.
   – $\mathcal{S}$ stores the output received from $\mathcal{S}_D$.

   If P_j is corrupted, then $\mathcal{S}$ performs the following steps for each honest party P_i.
   i. $\mathcal{S}$ sends c_i returned by $\mathcal{S}_D$ to $\mathcal{A}$.
   ii. $\mathcal{S}$ acts as the prover with $\mathcal{A}$ in the Σ-protocol.
   iii. $\mathcal{S}$ performs nothing (P_i will not fail in the above Σ-protocol, so this step will not be executed).

   If P_j is honest, then $\mathcal{S}$ performs the following steps for each corrupted party P_i.
   i. $\mathcal{S}$ receives $c_{i,j}$ from $\mathcal{A}$.
   ii. $\mathcal{S}$ acts as the verifier with $\mathcal{A}$ in the Σ-protocol.
   iii. If P_i fails the proof, then $\mathcal{S}$ performs the following steps.
      A. $\mathcal{S}$ broadcasts $corrupted_i$.
      B. For each honest party $P_k, k \neq j, \mathcal{S}$ sends $f_k(i)$ and $r_{k,i}$ to $\mathcal{A}$.

(b) $\mathcal{S}$ sets

$$S_{z_{\delta,j}} = msg_{\delta,j}.$$

   Modification upon corruption: If P_j gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 6, for P_j. Then $\mathcal{S}$ computes

$$val_\delta = \left(\sum_{k\in[n]} S_{u_\delta,k}\right) \cdot \left(\sum_{k\in[n]} S_{v_\delta,k}\right).$$

   Then, $\mathcal{S}$ sets

$$S_{z_\delta,j} = val_\delta - \sum_{k\in[n]\setminus\{j\}} msg_{\delta,k}.$$

12. For each $j \in [n], \mathcal{S}$ computes

$$rs_{z_{\delta,j}} = Opener\left(PK, SK, S_{z_\delta,j}, cip_{\delta,j}\right).$$

Modification upon corruption: If P_j gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 11, for P_j. Then $\mathcal{S}$ sets

$$rs_{z_{\delta,j}} = Opener\left(PK, SK, S_{z_\delta,j}, cip_{\delta,j}\right).$$

Simulating the output generation stage

Let

$$xI_i = \begin{cases} x_i' \text{ if } P_i \text{ is corrupted,} \\ x_i \text{ if } P_i \text{ is honest.} \end{cases}$$

$\mathcal{S}$ sends the set $\{xI_i\}_{i\in[n]:P_i \text{ is corrupted}}$ as the set of inputs of the corrupted parties to the trusted party. Then $\mathcal{S}$ receives back the set

$$\{yO_i\}_{i\in[n]:P_i \text{ is corrupted}} = \{f_i\left(xI_1, \ldots, xI_n\right)\}_{i\in[n]:P_i \text{ is corrupted}}$$

of the outputs of the corrupted parties from the trusted party.

1. For each $k \in [n], \mathcal{S}$ performs the following actions.

Case 1: P_k is honest.

(a) Simulating the share receiving of P_k.
$\mathcal{S}$ receives $S'_{\gamma+k,i}$ and $rs'_{\gamma+k,i}$ from $\mathcal{A}_h$, for each corrupted party P_i.
Modification upon corruption: If an honest party P_j gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 12 of computation stage, for P_j.

(b) Simulating the verification step of output generation stage.
For each corrupted party $P_i, \mathcal{S}$ does the consistency check as an honest P_k would. If P_i fails the consistency test, then $\mathcal{S}$ sends *corrupted*$_i$ to the trusted party and halts.
Modification upon corruption: If an honest party P_j gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 12 of computation stage, for P_j.

(c) Simulating the output computation step.
This step is a local computation step of P_k, so $\mathcal{S}$ does nothing.
Modification upon corruption: If an honest party P_j gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 12 of computation stage, for P_j.

Case 2: P_k is corrupted.

(a) Simulating the share sending to P_k.

i. $\mathcal{S}$ randomly selects the set $\left\{S'_{\gamma+k,j}\right\}_{j\in[n]:P_j \text{ is honest}}$ satisfying

$$\sum_{j\in[n]:P_j \text{ is honest}} S'_{\gamma+k,j} = yO_k - \sum_{i\in[n]:P_i \text{ is corrupted}} S_{\gamma+k,i}.$$

Modification upon corruption: If an honest party P_j gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 12 of computation stage, for P_j.

ii. For each honest party $P_i, \mathcal{S}$ performs the following steps.
– $\mathcal{S}$ computes

$$rs'_{\gamma+k,i} = \mathit{Opener}(PK, S'_{\gamma+k,i}, ES_{\gamma+k,k,i}).$$

– $\mathcal{S}$ sends $S'_{\gamma+k,i}$ as the share of P_i and $rs'_{\gamma+k,i}$ as the randomness of P_i to $\mathcal{A}_h$.

Modification upon corruption: If an honest party P_j gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 12 of computation stage, for P_j.

(b) Simulating the verification step of output generation stage.
If P_k broadcasts *corrupted*$_i$, then $\mathcal{S}$ sends *corrupted*$_i$ to the trusted party and halts.
Modification upon corruption: If an honest party P_j gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 12 of computation stage, for P_j.

(c) Simulating the output computation step.
This step is a local computation step of P_k, so $\mathcal{S}$ does nothing.
Modification upon corruption: If an honest party P_j gets corrupted after this step, then $\mathcal{S}$ performs the modification upon corruption steps listed in step 12 of computation stage, for P_j.

$\mathcal{S}$ outputs whatever $\mathcal{A}_h$ outputs.

6.2 The Detailed Proof

An *execution* of a protocol (either in the ideal world or in the hybrid world) is the process of running the protocol with a given adversary on given inputs, random inputs, and auxiliary input for the adversary. In the hybrid world an execution also depends on some extra inputs – the random choices of the trusted parties for evaluating the functionalities corresponding to the subprotocols.

Let the *internal history* of a party P_i at round ℓ be the concatenation of all the internal states from the beginning of the execution through round ℓ. The *global state* at round ℓ is defined as the concatenation of the internal histories of all the honest parties, the internal state of the adversary and the local state of the environment at round ℓ. This convention is used so that the global state of an execution at any round uniquely determines the continuation of the execution until its completion.

The global state is extended to rounds after the execution of the protocol has been completed, until the environment halts.

Proof. Let $\overline{x}$ be a balanced vector. Let $I \subset [n]$ denote the set of corrupted parties.

The view of the adversary at a given round of the execution is defined as the messages that the adversary receives at that round.

Let $ADVH_{Circuit,\mathcal{A}_h,I}(\ell, s, \overline{x}, z)$ denote the probability distribution of the view of the adversary $\mathcal{A}_h$ at round ℓ given the outputs of the honest parties, in the execution of protocol *Circuit* in the hybrid world on input $\overline{x} = x_1, \ldots, x_n$, security parameter s, auxiliary input z and corruption set I where the random inputs of the parties, the random input of the adversary and the random inputs of the trusted parties of the ideal functionalities are chosen uniformly at random from the corresponding domains.

Let $ADVI_{f,\mathcal{S},I}(\ell, s, \overline{x}, z)$ denote the probability distribution of the view of the simulated adversary $\mathcal{S}$ at round ℓ given the outputs of the honest parties, in the evaluation of functionality f in the ideal world on input $\overline{x} = x_1, \ldots, x_n$, security parameter s, auxiliary input z and corruption set I where the random inputs of the parties, the random input of the adversary and the random inputs of the simulator S for simulating the evaluations of the ideal functionalities are chosen uniformly at random from the corresponding domains.

First we describe the case where the input x_i of P_i for f is $abort_i$. The honest parties abort in the hybrid world in this case. In the ideal world, $\mathcal{S}$ sends $abort_i$ to the trusted party and halts. The trusted party sends $abort_i$ to all honest parties and halts. The honest parties abort. The output or view of the adversary is empty string in both worlds in this case, so (**??**) holds.

Then we describe the case where the input x_i of P_i for f is $corrupted_i$. The honest parties abort in the hybrid world in this case. In the ideal world, $\mathcal{S}$ sends $corrupted_i$ to the trusted party and halts. The trusted party sends $corrupted_i$ to all honest parties and halts. The honest parties abort. The view of the adversary is empty string in both worlds in this case, so (**??**) holds.

Next we describe the case where the input x_i of P_i for f is $cheat_i$. In the ideal world, $\mathcal{S}$ sends $cheat_i$ to the trusted party.

If the trusted party replies with $corrupted_i$ to $\mathcal{S}$, then $\mathcal{S}$ halts. The honest parties abort. In this case, the view of the adversary is empty string.

If the trusted party replies with *undetected* and the set $\{x_i\}_{i\in[n]\setminus I}$ of inputs of the honest parties for f to $\mathcal{S}$, then $\mathcal{S}$ sends *undetected* and the set $\{x_i\}_{i\in[n]\setminus I}$ to $\mathcal{A}_h$. Then $\mathcal{S}$ receives a vector $\{y'_i\}_{i\in[n]\setminus I}$ of outputs for the honest parties of the adversary's choice from $\mathcal{A}_h$. $\mathcal{S}$ sends this vector $\{y'_i\}_{i\in[n]\setminus I}$ to the trusted party and halts. For each honest party P_i, the trusted party sends y'_i as its output to P_i. In this case, the view of the adversary is the set $\{x_i\}_{i\in[n]\setminus I}$.

In the hybrid world, the trusted party receives the input $x_i = cheat_i$ from $\mathcal{A}_h$.

If the trusted party replies with $corrupted_i$ to the honest parties and halts, then the honest parties abort. In this case, the view of the adversary is empty string.

If the trusted party replies with *undetected* and the set $\{x_i\}_{i\in[n]\setminus I}$ of inputs of the honest parties for f to $\mathcal{A}_h$, then A_h sends the same vector $\{y'_i\}_{i\in[n]\setminus I}$ of outputs for the honest parties of the adversary's choice to the trusted party. For each honest party P_i, the trusted party sends y'_i as its output to P_i. In this case, the view of the adversary is the set $\{x_i\}_{i\in[n]\setminus I}$.

In both worlds, the view of the adversary is an empty string with probability $\epsilon = \frac{1}{2}$ and the set $\{x_i\}_{i\in[n]\setminus I}$ with probability $1-\epsilon = \frac{1}{2}$. So the output of the adversary, given the outputs of the honest parties, is identically distributed in two worlds in this case.

Next we describe the case where $\mathcal{S}$ sends $abort_i$ to the trusted party at some step of the simulation. $\mathcal{S}$ does this only if one of the following two situations arises.

1. the input of P_i for f_{CF} or f_{CC} or f_{OC} is $abort_i$.

2. $\mathcal{S}$ does not receive any message from $\mathcal{A}_h$ on behalf of a corrupted party P_i in some step although P_i is supposed to send some message in that step.

According to protocol *Circuit*, the honest parties abort in the hybrid world in both cases. In the ideal world, $\mathcal{S}$ sends $abort_i$ to the trusted party and halts. The trusted party sends $abort_i$ to all honest parties and halts. The honest parties abort. The output of $\mathcal{A}_h$ is the partial view of $\mathcal{A}_h$ up to the point of abort. Below we will prove that the distribution of the internal state of adversary $\mathcal{A}_h$ up to the end of the simulation, given the outputs of the honest parties in two worlds are computationally indistinguishable. Therefore, the distribution of the partial view of $\mathcal{A}_h$ up to the point of abort given the outputs of the honest parties in two worlds are also computationally indistinguishable. Then (**??**) holds for this case.

Next we consider the case where a party P_i gets caught cheating during the evaluation of f_{CF} or f_{CC} or f_{OC}. In this case, the cheating party gets caught in both worlds. The honest parties abort in both worlds. Similar to the case of abort, the output of $\mathcal{A}_h$ is the partial view of $\mathcal{A}_h$ up to the point P_i gets caught and we can prove that (**??**) holds.

For the remaining cases, we compare the view of the adversary given the outputs of the honest parties in two worlds, step by step as described below.

Step I: The CRS Generation Stage.

At round 1, parties call the trusted party T_{CF} of f_{CF} in the hybrid world.
In the hybrid world, T_{CF} of f_{CF} returns a string σ of length $p_1(\kappa)$. Here σ is distributed uniformly in the set of strings of length $p_1(s)$. In the ideal world, $\mathcal{S}$ sends a string σ_κ of length $p_1(\kappa)$ to $\mathcal{A}_h$. Distribution of σ and σ_κ are computationally indistinguishable.
Then we have

$$ADVI_{f,\mathcal{S},I}\left(1,s,\overline{x},z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(1,s,\overline{x},z\right). \tag{2}$$

Step II: The Key Generation Stage.

Let ℓ_{KeyGen} and $\ell_{KeyGenEnd}$ denote the round at which the parties start and end the key generation stage, respectively.
At step $2(d)$ of the simulation, $\mathcal{S}$ checks whether $d_{key} = m_{key}$. If $d_{key} \neq m_{key}$, then $\mathcal{S}$ rewinds back to step $2(b)$ of the simulation. Since d_{key} is selected uniformly at random from $\{1,2\}$, the expected number of rewinds until $d_{key} = m_{key}$ is at most 2. That means the execution of steps 2(b)-2(d) of $\mathcal{S}$ needs expected constant time. To bound the running time of these steps within a polynomial of the security parameter s, we can continue rewinding $\mathcal{S}$ at most s^ℓ times where $\ell \in \mathbb{N}$. If $d_{key} \neq m_{key}$ after s^ℓ rewinds, then $\mathcal{S}$ fails. The probability of failure of $\mathcal{S}$ is negligible.
We consider the three cases described in step 2(e). In any of the cases, $\mathcal{S}$ has to send $v_{i,m_{key}}$ to $\mathcal{A}_h$ for each honest party P_i, only after v generates d_{key} that equals the challenge m_{key}. $\mathcal{S}$ computes the key pair $(u_{i,d_{key}}, v_{i,d_{key}})$ in the injective mode, so none of the honest parties will be caught cheating in step 2(e).

Case 1: $\mathcal{A}_h$ does not send any lossy key in any of the simulations.

The message that $\mathcal{A}_h$ receives during the key generation stage, given the outputs of the honest parties, consists of the following elements.

1. The set of public keys $\{u_{i,1}, u_{i,2}\}_{i\in[n]\setminus I}$ that $\mathcal{A}_h$ receives at step 2(b).
   In the hybrid world, each honest party P_i sends two injective public keys.
   In the ideal world, for each honest party P_i, $\mathcal{S}$ sends one injective public key $u_{i,d_{key}}$ and one lossy public key $u_{i,3-d_{key}}$. By "the indistinguishability of keys" property of a lossy encryption scheme, a lossy public key and an injective public key are computationally indistinguishable.
   Therefore the distributions of these public keys in two worlds are computationally indistinguishable.
2. The set of private keys $\{v_{i,m_{key}}\}_{i\in[n]\setminus I}$ that $\mathcal{A}_h$ receives at step 2(d).
   Note that $\mathcal{S}$ reaches this point of simulation only when $d_{key} = m_{key}$. Since the key pair $(u_{i,m_{key}}, v_{i,m_{key}})$ is generated in the injective mode in both worlds for each honest party P_i, the distributions of $\{v_{i,m_{key}}\}_{i\in[n]\setminus I}$ are identical in two worlds.

Case 2: $\mathcal{A}_h$ sends lossy keys for a party P_i in exactly one of the simulations (corresponding to challenge m_{ck}).

If $m_{ck} = m_{key}$, then the lossy pair is opened for verification. $m_{ck} = m_{key}$ with probability $\frac{1}{2}$. In the hybrid world, the honest parties abort. In the ideal world, $\mathcal{S}$ sends $corrupted_i$ to the trusted party and later the honest parties abort. The output of $\mathcal{A}_h$ is its partial view up to the point the protocol terminates and we can prove indistinguishablity.

If $m_{ck} \neq m_{key}$, then the injective key is opened. $m_{ck} \neq m_{key}$ with probability $\frac{1}{2}$. So P_i does not get caught in both worlds. The key pair (PK_i, SK_i) is a lossy pair of keys. The execution continues to the next step in this case. In this case, the messages that the adversary receives in this stage is distributed identically to case 1.

Case 3: $\mathcal{A}_h$ sends lossy keys for a party P_i in both simulations.

In the hybrid world, the honest parties catches cheating P_i with probability 1 in this case since whatever the challenge be, P_i gets caught. The honest parties can detect cheating by the "key pair detection" property of the encryption scheme. The honest parties abort.

In the ideal world, $\mathcal{S}$ sends $corrupted_i$ to the trusted party. The honest parties abort. In particular, this means that the simulator $\mathcal{S}$ can catch a cheating party P_i with probability 1 when $\mathcal{A}_h$ sends a lossy pair of keys on behalf of P_i in both simulations. This holds by the "key pair detection" property of the encryption scheme.

The output of $\mathcal{A}_h$ is its partial view up to the point the protocol terminates. We can prove indistinguishablity by case 1.

From (2) and the analysis presented above, we can say that the following holds:

$$ADVI_{f,\mathcal{S},I}\left(\ell_{KeyGenEnd}, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{KeyGen}, s, \overline{x}, z\right). \quad (3)$$

Step III: The Input Sharing Stage.

Let $\ell_{InShare}$ and $\ell_{InShareEnd}$ denote the round at which the parties start and end the input sharing stage, respectively.

We consider the three cases described in step 3(f). $\mathcal{S}$ computes the enryptions honestly, so none of the honest parties will be caught cheating in step 3(f).

Case 1: $\mathcal{A}_h$ does not send any invalid response in any of the simulations.

In the ideal world, for each $i \in I, \mathcal{S}$ computes the shares $B_{1,i,i}$ and $B_{2,i,i}$ of P_i in two simulations, by decrypting $Y_{1,i,i}$ and $Y_{2,i,i}$.

$\mathcal{S}$ computes x'_i where x'_i is the substituted input of P_i by $\mathcal{A}_h$.

The message that $\mathcal{A}_h$ receives during the input sharing stage, given the outputs of the honest parties, consists of the following elements.

1. The set of ciphertexts $\{Y_{\ell,i,j}\}_{i\in[n]\setminus I,\ell\in\{1,2\},j\in[n]}$ that $\mathcal{A}_h$ receives at step 3(b).
   In the hybrid world, each honest party P_i sends encryptions of shares of its actual input x_i.
   In the ideal world, $\mathcal{S}$ sends encryptions of shares of zero.
   Note that $\mathcal{A}_h$ knows the private keys of the corrupted parties. So $\mathcal{A}_h$ can decrypt the ciphertexts generated under the public keys of the corrupted parties and thereby compute the set $\{B_{\ell,i,j}\}_{i\in[n]\setminus I,\ell\in\{1,2\},j\in I}$ of the shares of the corrupted parties. Since there is at least one honest party and the input x_i of P_i is unknown to $\mathcal{A}_h$, the set of decrypted shares of the corrupted parties in two worlds are identically distributed.
   $\mathcal{A}_h$ does not know the private keys of the honest parties. By the semantic security of the encryption scheme, for each $i \in [n] \setminus I$, the ciphertexts generated under the public key PK_i in two worlds are computationally indistinguishable.
2. The set of shares $\{B_{1,i,j}, B_{2,i,j}\}_{i\in[n]\setminus I,\ell\in\{1,2\},j\in I}$ that $\mathcal{A}_h$ receives at step 3(c).
   In the hybrid world, each honest party P_i sends the shares of its actual input x_i.
   In the ideal world, $\mathcal{S}$ sends the shares of zero.
   Since there is at least one honest party and the input x_i of P_i is unknown to $\mathcal{A}_h$, it holds that the set of shares of the corrupted parties in two worlds are identically distributed.
3. The set of randomness $\{b_{1,i,j}, b_{2,i,j}\}_{i\in[n]\setminus I,\ell\in\{1,2\},j\in I}$ that $\mathcal{A}_h$ receives at step 3(c).
   These strings are generated uniformly at random in both worlds.

4. The set of shares $\{B_{m_{in},i,j}\}_{i\in[n]\setminus I, j\in[n]\setminus\{i\}}$ that $\mathcal{A}_h$ receives at step 3(e).
   In the hybrid world, each honest party P_i sends $(n-1)$ shares of its actual input x_i.
   In the ideal world, $\mathcal{S}$ sends $(n-1)$ shares of zero.
   Since the input x_i of P_i is unknown to $\mathcal{A}_h$ and there are n additive shares, the set of these $(n-1)$ shares in two worlds are identically distributed.
5. The set of randomness $\{b_{m_{in},i,j}\}_{i\in[n]\setminus I, j\in[n]\setminus\{i\}}$ that $\mathcal{A}_h$ receives at step 3(e).
   These strings are generated uniformly at random in both worlds.

Case 2: $\mathcal{A}_h$ sends invalid response in exactly one of the simulations (corresponding to the challenge m_{ci}).
If $m_{ci} = m_{in}$, then P_i gets caught in both worlds. $m_{ci} = m_{in}$ with probability $\frac{1}{2}$. In the hybrid world, the honest parties abort. In the ideal world, $\mathcal{S}$ sends *corrupted*$_i$ to the trusted party and later the honest parties abort. The output of $\mathcal{A}_h$ is its partial view up to the point the protocol terminates. So we can prove indistinguishability by case 1.
If $m_{ci} \neq m_{in}$, then P_i does not get caught in both worlds. $m_{ci} \neq m_{in}$ with probability $\frac{1}{2}$. The execution continues to the next step in this case. Indistinguishability can be proved similar to case 1.

Case 3: $\mathcal{A}_h$ sends invalid responses for a party P_i in both simulations.
In the hybrid world, the honest parties catch cheating P_i with probability 1 in this case since whatever the challenge be, P_i gets caught. The honest parties abort.
In the ideal world, $\mathcal{S}$ sends *corrupted*$_i$ to the trusted party. The honest parties abort.
The output of $\mathcal{A}_h$ is its partial view up to the point the protocol terminates and we can prove indistinguishability by case 1.

From (3) and the above analysis, we have

$$ADVI_{f,\mathcal{S},I}\left(\ell_{InShareEnd}, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{InShareEnd}, s, \overline{x}, z\right). \tag{4}$$

Step IV: The Computation Stage.

In the computation stage, only the multiplication gates need interaction among the parties. The local computations for gates other than multiplication gates can be performed in the same round at which a multiplication gate is evaluated. To simplify our analysis, we assume that the computation of each gate is performed in a separate round.
Let ℓ_δ denote the round at which parties start the evaluation of gate g_δ. Then $\ell_{\delta+1}-1$ denote the round at which the evaluation of gate g_δ is finished.
The evaluation of addition gates and multiplication-by-constant gates are deterministic process and do not need any interaction among parties.The process of the simulation of the evaluation of these two types of gates by $\mathcal{S}$ is also deterministic.
We will prove the following by induction on δ.

$$ADVI_{f,\mathcal{S},I}\left(\ell_{\delta+1}-1, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{\delta+1}-1, s, \overline{x}, z\right).$$

The proof for the base case and the induction case is similar – we describe only the induction case.
We assume that the induction hypothesis holds for the evaluation of gate $g_{\delta'}$ for any $\delta' < \delta$.
First we describe the comparison of the views of the adversary in two worlds, at the round at which the shares were evaluated for the input wires of gate g_δ.
Let ℓ_{u_δ} and ℓ_{v_δ} denotes the round at which the shares associated with the input wire u_δ and v_δ of gate g_δ were evaluated, respectively.
The evaluation of shares associated with the wire w_{u_δ} may have been done

1. either in the input sharing stage. This happens if $u_\delta \leq n$, that is, if w_{u_δ} is an input wire of C. In this case, $\ell_{u_\delta} = \ell_{InShareEnd}$. By stage III, we have

$$ADVI_{f,\mathcal{S},I}\left(\ell_{u_\delta}, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{u_\delta}, s, \overline{x}, z\right).$$

2. or in the computation stage. This happens if $u_\delta > n$, that is, if w_{u_δ} is the output wire of another gate g_{δ_1}. In this case, $\ell_{u_\delta} = \ell_{\delta_1+1}-1$. Since the gates of C are evaluated according to the topological

ordering $g_1, \ldots, g_\theta$, it holds that $\delta_1 < \delta$. Then the induction hypothesis holds for the evaluation of gate g_{δ_1}. So we have

$$\begin{aligned} ADVI_{f,\mathcal{S},I}\left(\ell_{\delta_1+1}-1, s, \overline{x}, z\right) &\overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{\delta_1+1}-1, s, \overline{x}, z\right) \\ \Leftrightarrow ADVI_{f,\mathcal{S},I}\left(\ell_{u_\delta}, s, \overline{x}, z\right) &\overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{u_\delta}, s, \overline{x}, z\right). \end{aligned}$$

Similarly, for the input wire v_δ we have

$$ADVI_{f,\mathcal{S},I}\left(\ell_{v_\delta}, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{v_\delta}, s, \overline{x}, z\right).$$

There are three possible cases according to the type of gate g_δ, as follows.

Case 1: g_δ is an addition gate.

In the hybrid world, each party is supposed to set its shares $S_{z_\delta,i}, rs_{z_\delta,i}$ and $\{ES_{z_\delta,i,j}\}_{j\in[n]\setminus\{i\}}$ associated with the output wire w_{z_δ} as a deterministic function of its shares associated with the input wires w_{u_δ} and w_{v_δ}.

In the ideal world, for each party P_i, the simulator $\mathcal{S}$ computes the shares of P_i for the output wire w_{z_δ} just as an honest P_i would.

The view of adversary at round $(\ell_{\delta+1}-1)$ is a deterministic function of its view at round ℓ_{u_δ} and round ℓ_{v_δ}. From the analysis of views in two worlds at round ℓ_{u_δ} and at round ℓ_{v_δ}, we can say that the view of the adversary at round $(\ell_{\delta+1}-1)$ given the outputs of the honest parties are computationally indistinguishable in two worlds, that is,

$$ADVI_{f,\mathcal{S},I}\left(\ell_{\delta+1}-1, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{\delta+1}-1, s, \overline{x}, z\right).$$

Case 2: g_δ is a multiplication-by-constant gate.

The analysis is similar to case 1 – the only difference is that we have a single input in this type of gates.

Case 3: g_δ is a multiplication gate.

In this case, the parties evaluate the multiplication gate by perfroming a series of steps.

At step $4(h)$ of the simulation, $\mathcal{S}$ checks whether $d = m$. If $d \neq m$, then $\mathcal{S}$ rewinds back to step $4(c)$ of the simulation. Like the key generation stage, we can prove that steps 4(c)-4(h) can be performed in polynomial time with a negligible probability of failure.

$\mathcal{S}$ has to send

$$\left\{A_{i,m}, B_{i,m}, r_{i,m}, v_{i,m,1}, v_{i,m,2}, \{H_{i,j,m,\ell}, g_{i,j,m,\ell}, u_{i,j,m,\ell}\}_{j\in[n]\setminus\{i\},\ell\in\{1,2\}}\right\}$$

to $\mathcal{A}_h$, for each honest party P_i, only after $\mathcal{S}$ generates d that equals the challenge m. S computes these values as an honest P_i would, so none of the honest parties will be caught cheating in step 4(i). Now we consider the three cases described in step 4(i).

Case 1 : $\mathcal{A}_h$ does not send any invalid response in any of the simulations.

The view of $\mathcal{A}_h$, given the outputs of the honest parties, consists of the following elements.

1. The set of ciphertexts $\{X_{i,m}, Y_{i,m}\}_{i\in[n]\setminus I}$ that $\mathcal{A}_h$ receives in step 4(c)(D).
   In both worlds, these ciphertexts are generated according to the protocol specification. In particular, the plaintexts are selected uniformly at random and then encrypted.
   $\mathcal{A}_h$ does not know the private keys of the honest parties. By the semantic security of the encryption scheme, for each honest party P_i it holds that the ciphertexts under PK_i in two worlds are computationlly indistinguishable.
2. The set of ciphertexts $\{X_{i,3-m}, Y_{i,3-m}\}_{i\in[n]\setminus I}$ that $\mathcal{A}_h$ receives in step 4(c)(D).
   In the hybrid world, each honest party P_i computes these ciphertexts according to the protocol.
   In the ideal world, for each honest party P_i, $\mathcal{S}$ computes $X_{i,3-m}, Y_{i,3-m}$ as follows.

$$X_{i,3-m} = EA_{q,i} -_h X_{i,m} -_h E_{PK_i}(0, a_{i,0}) \text{ where } q \in I.$$

$$Y_{i,3-m} = EB_{q,i} -_h Y_{i,m} -_h E_{PK_i}(0, b_{i,0}) \text{ where } q \in I.$$

For each honest party P_i, $X_{i,3-m}$ and $Y_{i,3-m}$ are encryptions under PK_i.

$\mathcal{A}_h$ does not know the private keys of the honest parties. By the semantic security of the encryption scheme, for each honest party P_i it holds that the ciphertexts under PK_i in the ideal world are computationlly indistinguishable from the ciphertexts under PK_i in the hybrid world.

3. The set of ciphertexts $\{G_{i,j,m,\ell}\}_{i\in[n]\setminus I, j\in[n]\setminus\{i\},\ell\in\{1,2\}}$ that $\mathcal{A}_h$ receives in step 4(c)(F).
   In both worlds, these ciphertexts are generated according to the protocol specification. In particular, the plaintexts are selected uniformly at random and then encrypted.
   $\mathcal{A}_h$ does not know the private keys of the honest parties. By the semantic security of the encryption scheme, for each honest party P_i it holds that the ciphertexts under PK_i in two worlds are computationlly indistinguishable.
4. The set of ciphertexts $\{G_{i,j,3-m,\ell}\}_{i\in[n]\setminus I, j\in[n]\setminus\{i\},\ell\in\{1,2\}}$ that $\mathcal{A}_h$ receives in step 4(c)(F).
   In the hybrid world, each honest party P_i computes these ciphertexts according to the protocol.
   In the ideal world, for each honest party P_i, $\mathcal{S}$ chooses a random field element $H_{i,j,3-m,1}$, two random strings $g_{i,j,3-m,1}, c_{i,j,0}$, then computes and sends the following to $\mathcal{A}_h$.
$$G_{i,j,3-m,1} = E_{PK_i}(H_{i,j,3-m,1}, g_{i,j,3-m,1})$$
$$G_{i,j,3-m,2} = EC_{q,i,j} -_h \sum_{\ell\in\{1,2\}} G_{i,j,m,\ell} -_h G_{i,j,3-m,1} -_h E_{PK_i}(0, c_{i,j,0}) \text{ where } q \in I.$$
   For each honest party P_i, the ciphertexts $\{G_{i,j,3-m,1}, G_{i,j,3-m,2}\}_{j\in[n]\setminus\{i\}}$ are encryptions under PK_i.
   $\mathcal{A}_h$ does not know the private keys of the honest parties. By the semantic security of the encryption scheme, for each honest party P_i it holds that the ciphertexts under PK_i in the ideal world are computationlly indistinguishable from the ciphertexts under PK_i in the hybrid world.
5. The set of random strings $\left\{a_{i,0}, b_{i,0}, \{c_{i,j,0}\}_{j\in[n]\setminus\{i\}}\right\}_{i\in[n]\setminus I}$ that $\mathcal{A}_h$ receives at step 4(d).
   In the hybrid world, each honest party P_i computes these strings according to the protocol.
   In the ideal world, $\mathcal{S}$ chooses these strings uniformly at random.
   In the hybrid world, the string $a_{i,0}$ is computed by the following equation.
$$\begin{aligned} E_{PK_i}(0, a_{i,0}) &= E_{PK_i}(A_i, ra_i) -_h X_{i,1} -_h X_{i,2} \\ &= E_{PK_i}(A_i, ra_i) -_h E_{PK_i}(A_{i,1}, r_{i,1}) -_h E_{PK_i}(A_{i,2}, r_{i,2}). \end{aligned}$$
   The value of $a_{i,0}$ depends on $E_{Pk_i}(A_i, ra_i), E_{PK_i}(A_{i,1}, r_{i,1})$ and $E_{PK_i}(A_{i,2}, r_{i,2})$. $\mathcal{A}_h$ only knows the ciphertext $EA_{q,i}$ which is supposed to be $E_{PK_i}(A_i, ra_i)$. So the distributions of the strings $a_{i,0}$ in two worlds are computationally indistinguishable. We can show similar result for the distributions of the strings $b_{i,0}$ and $\{c_{i,j,0}\}_{j\in[n]\setminus\{i\}}$.
6. The set of ciphertexts $\{L_{i,k,\ell}\}_{i\in[n]\setminus I,(k,\ell)\in\{1,2\}^2}$ that $\mathcal{A}_h$ receives at step 4(e).
   In the hybrid world, each honest party P_i computes these ciphertexts based on its actual shares A_i and B_i.
   In the ideal world, for each honest party P_i, $\mathcal{S}$ computes these ciphertexts based on fake shares of P_i that are obtained by using zero as the inputs for the honest parties.
   $\mathcal{A}_h$ does not know the private keys of the honest parties. By the semantic security of the encryption scheme, for each honest party P_i it holds that the ciphertexts under PK_i in the ideal world are computationlly indistinguishable from the ciphertexts under PK_i in the hybrid world.
7. The set of ciphertexts $\{K_{i,j,k,\ell}\}_{i\in[n]\setminus I, j\in[n]\setminus\{i\},(k,\ell)\in\{1,2\}^2}$ that $\mathcal{A}_h$ receives at step 2(f).
   In the hybrid world, each honest party P_i computes these ciphertexts based on its actual share A_i.
   In the ideal world, for each honest party P_i, $\mathcal{S}$ computes these ciphertexts based on fake shares of P_i that are obtained by using zero as the inputs for the honest parties.

$\mathcal{A}_h$ does not know the private keys of the honest parties. By the semantic security of the encryption scheme, for each honest party P_i it holds that the ciphertexts under PK_i in the ideal world are computationlly indistinguishable from the ciphertexts under PK_i in the hybrid world.

8. The message received by $\mathcal{A}_h$ at step 4(h), which consists of

$$\left\{A_{i,m}, B_{i,m}, r_{i,m}, v_{i,m,1}, v_{i,m,2}, \left\{H_{i,j,m,\ell}, g_{i,j,m,\ell}, u_{i,j,m,\ell}\right\}_{j\in[n]\setminus\{i\},\ell\in\{1,2\}}\right\}_{i\in[n]\setminus I}.$$

For each honest party P_i, it holds that exactly one subshare (namely, $A_{i,m}$ and $B_{i,m}$) out of the two additive subshares of its shares A_i and B_i are sent to $\mathcal{A}_h$. So the distributions of the opened shares are uniform in both worlds.
For each honest party P_i and each $C_{i,j}$ where $j \in [n]\setminus\{i\}$, it holds that exactly two subshares (namely, $\{H_{i,j,m,\ell}\}_{\ell\in\{1,2\}}$) out of four additive subshares of $C_{i,j}$ are sent to $\mathcal{A}_h$. Therefore the distributions of the opened shares are uniform in both worlds.
For each honest party P_i, the strings $r_{i,m}, v_{i,m,1}, v_{i,m,2}, \{g_{i,j,m,\ell}, u_{i,j,m,\ell}\}_{j\in[n]\setminus\{i\},\ell\in\{1,2\}}$ are generated uniformly at random in both worlds.
Therefore the message sent to $\mathcal{A}_h$ at step 4(h) are distributed identically in two worlds.

Case 2 : $\mathcal{A}_h$ sends invalid response in exactly one of the simulations (corresponding to the challenge m_{cm}).
If $m_{cm} = m$, then P_i gets caught in both worlds. $m_{cm} = m$ with probability $\frac{1}{2}$. In the hybrid world, the honest parties abort. In the ideal world, $\mathcal{S}$ sends *corrupted*$_i$ to the trusted party and later the honest parties abort. The output of $\mathcal{A}_h$ is its partial view up to the point the protocol terminates. By case 1, we can prove indistinguishability.
If $m_{ci} \neq m_{in}$, then P_i does not get caught in both worlds. $m_{cm} \neq m$ with probability $\frac{1}{2}$. The execution continues to the next step in this case.

Case 3 : $\mathcal{A}_h$ sends invalid responses for a party P_i in both simulations.
In the ideal world, $\mathcal{S}$ sends *corrupted*$_i$ to the trusted party. The honest parties abort.
In the real world, the honest parties would output *corrupted*$_i$ with probability 1 in this case since whatever the challenge be, P_i would get caught. The honest parties abort.
The output of $\mathcal{A}_h$ is its partial view up to the point the protocol terminates. By case 1, we can prove indistinguishability.

So we have

$$ADVI_{f,\mathcal{S},I}\left(\ell_{\delta+1}-1, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{\delta+1}-1, s, \overline{x}, z\right). \tag{5}$$

From (5) for $\delta = \theta$, we have

$$ADVI_{f,\mathcal{S},I}\left(\ell_{\theta+1}-1, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{\theta+1}-1, s, \overline{x}, z\right). \tag{6}$$

Step V: The Output Generation Stage.

Let ℓ_{Out} denote the round at which the parties start the output generation stage in the hybrid world. We see that one round is needed to coomunicate message for evaluating one output wire of the circuit. Then $\ell_{Out} + k - 1$ denotes the round at which the parties send message to P_k.
We will prove the following by induction on k.

$$ADVI_{f,\mathcal{S},I}\left(\ell_{Out}+k-1, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{Out}+k-1, s, \overline{x}, z\right). \tag{7}$$

The proof for the base case and the induction case is similar – we describe only the induction case. For the base case, from (6), we have

$$ADVI_{f,\mathcal{S},I}\left(\ell_{Out}-1, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{Out}-1, s, \overline{x}, z\right). \tag{8}$$

as $\ell_{Out} = \ell_{\theta+1}$. We assume that the induction hypothesis holds for the k'-th evaluation of f_{Out} for any $k' < k$. From $k' = k-1$, we have

$$ADVI_{f,\mathcal{S},I}\left(\ell_{Out}+k-2, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{Out}+k-2, s, \overline{x}, z\right). \tag{9}$$

Observe that

$$xI_i = \begin{cases} x'_i \text{ if } i \in I \\ x_i \text{ if } i \notin I. \end{cases}$$

where x'_i denotes the replaced input of the corrupted party P_i.
yO_k is the output of P_k obtained from the trusted party, after $\mathcal{S}$ sends the vector $\{xI_i\}_{i\in I}$ as the inputs of the corrupted parties to the trusted party. Each honest party P_i sends its actual input x_i to the trusted party, so it follows that

$$yO_k = f_k\left(xI_1, \ldots, xI_n\right).$$

Case 1: P_k is honest.
$\mathcal{S}$ does not send any message to $\mathcal{A}_h$ in this case.
If A_h does not send any message for some corrupted party P_i at step 5(a)(i), then $\mathcal{S}$ sends $abort_i$ to the trusted party. The trusted party sends $abort_i$ to all honest parties and halts. The honest parties abort. In the hybrid world, honest P_k broadcasts $abort_i$ and aborts. The remaining honest parties abort. The view of adversary is its view up to round $(\ell_{Out} + k - 2)$. By (8) and (9), the view of the adversary given the outputs of the honest parties are computationally indistinguishable in two worlds. The simulation ends here in this case.
In the case where no party aborts, there are two possible cases, as described below.
Case 1: $\mathcal{A}_h$ sends the correct share and randomness for each corrupted party in step 5(a)(i).
In this case, there has been no cheating from the corrupted parties. In the ideal world, honest P_k computes its output L_k correctly.
In the hybrid world, P_k gets the correct shares and randomness from all parties at step 5(a)(i). No party gets caught during verification at step 5(a)(ii). Since P_k is honest, it computes L_k correctly by summing the shares of all parties.
Case 2: $\mathcal{A}_h$ sends inconsistent share and randomness for some corrupted party P_i in step 5(a)(i).
Here we describe the situation where $\mathcal{A}_h$ sends inconsistent share and randomness for a single corrupted party P_i in step 5(a)(i). A similar analysis can be performed for the scenario where $\mathcal{A}_h$ sends inconsistent share and randomness for more than one corrupted parties in step 5(a)(i).
There are two possible outcomes, as described below.
Case 2(a): The key pair (PK_i, SK_i) of P_i is a lossy pair of keys.
This event can happen with probability at most $\frac{1}{2}$. Recall that one out of two pairs of keys of each party are checked during the key generation stage. Since P_i was not caught during that stage, it holds that P_i used one lossy pair and one injective pair in the key generation stage and the lossy pair was not opened. Since the challenge $m_{key} \in \{1, 2\}$ is generated uniformly at random, this situation can arise with probability at most $\frac{1}{2}$. In that case, P_i can send a different pair $(S'_{\gamma+k,i}, rs'_{\gamma+k}, i)$ of share and randomness for the wire $w_{\gamma+k}$ to P_k, but the cheating by P_i cannot be detected. This is due to the *openability* property of the lossy encryption scheme.
In the ideal world, the simulation continues as in case 1. Honest P_k computes an incorrect output

$$L'_k = L_k - S_{\gamma+k,i} + S'_{\gamma+k,i}.$$

In the hybrid world, honest P_k cannot catch the cheating party P_i in the verification of step 5(a)(ii). P_k computes the incorrect output L'_k.
Case 2(b): The key pair (PK_i, SK_i) of P_i is an injective pair of keys.
The probability of this event is at most $\frac{1}{2}$.
In this case, P_i gets caught in both worlds. In the ideal world, $\mathcal{S}$ checks consistency of inputs at step 5(a)(ii) and catches cheating P_i. Then $\mathcal{S}$ sends $corrupted_i$ to the trusted party and the trusted party sends $corrupted_i$ to all honest parties and halts. The honest parties abort.
In the hybrid world, honest P_k catches cheating P_i during verification of step 5(a)(ii). Then P_k broadcasts $corrupted_i$ and aborts. The honest parties abort.
Case 2: P_k is corrupted.
Observe that, in the key generation stage, at step 2(f), $\mathcal{S}$ sets

$$(PK_i, SK_i) = \left(u_{i,3-m_{key}}, v_{i,3-m_{key}}\right) \text{ for each } i \in [n].$$

This implies that

$$(PK_i, SK_i) \in G(1^s, lossy)$$

for each $i \in [n] \setminus I$, in the ideal world.
By the existence of an efficient *Opener* algorithm, $\mathcal{S}$ can compute $rs'_{\gamma+k,i}$ for each honest party P_i in polynomial time in step 5(a)(ii). The probability of failure to compute such an rs'_i is negligible. By the openability property of a lossy encryption scheme,

$$E_{PK_i}(S'_{\gamma+k,i}, rs'_{\gamma+k,i}) = ES_{\gamma+k,k,i}$$

for each honest party P_i. So no honest party will get caught cheating during the verification of step 5(a)(ii).
The honest parties output empty strings in both worlds.
In the hybrid world, each honest party P_i sends its actual share $S_{\gamma+k,i}$ and its actual randomness $rs_{\gamma+k,i}$ for wire $w_{\gamma+k,i}$ to P_k at step 5(a)(i).
In the ideal world, $\mathcal{S}$ randomly selects the set of shares $\left\{S'_{\gamma+k,i}\right\}_{i\in[n]\setminus I}$ satisfying

$$\sum_{i\in[n]\setminus I} S'_{\gamma+k,i} = YO_k - \sum_{j\in I} S_{\gamma+k,j}.$$

Since both of these sets sum to $\left(YO_k - \sum_{j\in I} S_{\gamma+k,j}\right)$, it holds that the set of shares $\{S_{\gamma+k,i}\}_{i\in[n]\setminus I}$ and $\{S'_{\gamma+k,i}\}_{i\in[n]\setminus I}$ in two worlds are identically distributed.
Observe that the adversary knows the set $\{ES_{\gamma+k,k,i}\}_{i\in[n]\setminus I}$ of encrypted shares of the honest parties for wire $w_{\gamma+k,i}$, as part of the inputs of the corrupted party P_k.
In the ideal world, $\mathcal{S}$ sends the computed fake string $rs'_{\gamma+k,i}$ for each honest party P_i such that

$$E_{PK_i}(S'_{\gamma+k,i}, rs'_{\gamma+k,i}) = ES_{\gamma+k,k,i}.$$

Since the ciphertexts match, the adversary does not detect any inconsistency for any of the honest parties.
$rs'_{\gamma+k,i}$ is computed as follows.

$$rs'_{\gamma+k,i} = Opener(PK_i, S'_{\gamma+k,i}, ES_{\gamma+k,k,i}).$$

The value of the string $rs'_{\gamma+k,i}$ depends on $PK_i, ES_{\gamma+k,k,i}$ and $S'_{\gamma+k,k,i}$. PK_i and $ES_{\gamma+k,k,i}$ is known to $\mathcal{A}_h$ but $S'_{\gamma+k,i}$ is selected at random by S. Therefore the distributions of the strings $\{rs_{\gamma+k,i}\}_{i\in[n]\setminus I}$ in the hybrid world and $\{rs'_{\gamma+k,i}\}_{i\in[n]\setminus I}$ in the ideal world are computationally indistinguishable.

In any case, we have proved that

$$ADVI_{f,\mathcal{S},I}\left(\ell_{Out}+k-1, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{Out}+k-1, s, \overline{x}, z\right). \tag{10}$$

From (10) for $k = n$, we have

$$ADVI_{f,\mathcal{S},I}\left(\ell_{Out}+n-1, s, \overline{x}, z\right) \overset{c}{\equiv} ADVH_{Circuit,\mathcal{A}_h,I}\left(\ell_{Out}+n-1, s, \overline{x}, z\right). \tag{11}$$

From (11), we have

$$\begin{aligned}&\left\{IDA_{f,\mathcal{S}(z),I}(\overline{x}, s) | IDH_{f,\mathcal{S}(z),I}(\overline{x}, s)\right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}}\\ \overset{c}{\equiv}&\left\{HYA^{(f_{CF},1),(f_{CC},1),(f_{OC},1)}_{Circuit,\mathcal{A}_h(z),I}(\overline{x}, s) | HYH^{(f_{CF},1),(f_{CC},1),(f_{OC},1)}_{Circuit,\mathcal{A}_h(z),I}(\overline{x}, s)\right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}}\end{aligned}$$

6.3 Proof of Theorem 2

Proof. Let $\epsilon = \frac{1}{2}$. Let $\mathcal{A}_h$ be a static covert adversary that interacts with parties running protocol *Circuit* in the $((f_{CF},1),(f_{CC},1),(f_{OC},1))$-hybrid world.

Let $\overline{x}$ be a balanced vector. Let $I \subset [n]$ denote the set of corrupted parties.

By Lemma **??**,

$$\left\{IDH_{f,\mathcal{S}(z),I}(\overline{x},s)\right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}} = \left\{HYH^{(f_{CF},1),(f_{CC},1),(f_{OC},1)}_{Circuit,\mathcal{A}_h(z),I}(\overline{x},s)\right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}} \tag{12}$$

By Lemma **??**,

$$\begin{aligned}&\left\{IDA_{f,\mathcal{S}(z),I}(\overline{x},s)|IDH_{f,\mathcal{S}(z),I}(\overline{x},s)\right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}}\\ \overset{c}{\equiv}&\left\{HYA^{(f_{CF},1),(f_{CC},1),(f_{OC},1)}_{Circuit,\mathcal{A}_h(z),I}(\overline{x},s)|HYH^{(f_{CF},1),(f_{CC},1),(f_{OC},1)}_{Circuit,\mathcal{A}_h(z),I}(\overline{x},s)\right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}}\end{aligned} \tag{13}$$

From (12) and (13), we have

$$\begin{aligned}&\left\{IDH_{f,\mathcal{S}(z),I}(\overline{x},s),IDA_{f,\mathcal{S}(z),I}(\overline{x},s)\right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}}\\ \overset{c}{\equiv}&\left\{HYH^{(f_{CF},1),(f_{CC},1),(f_{OC},1)}_{Circuit,\mathcal{A}_h(z),I}(\overline{x},s),HYA^{(f_{CF},1),(f_{CC},1),(f_{OC},1)}_{Circuit,\mathcal{A}_h(z),I}(\overline{x},s)\right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}},\end{aligned}$$

that is,

$$\left\{IDEAL^{\epsilon}_{f,\mathcal{S}(z),I}(\overline{x},s)\right\}_{\overline{x},z\in(\{0,1\}^*)^{n+1};s\in\mathbb{N}} \overset{c}{\equiv} HYBRID^{(f_{CF},1),(f_{CC},1),(f_{OC},1)}_{Circuit,\mathcal{A}(z),I}(\overline{x},s).$$

References

AL10. Yonatan Aumann and Yehuda Lindell. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. *Journal of Cryptology*, 23(2):281–343, 2010.

BHY09. Mihir Bellare, Dennis Hofheinz, and Scott Yilek. Possibility and Impossibility Results for Encryption and Commitment Secure under Selective Opening. In *Proceedings, Advances in Cryptology — EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 1–35, Berlin, Heidelberg, 2009. Springer.

Can00. Ran Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

Can01. Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Proceedings, 42nd IEEE Symposium on Foundations of Computer Science — FOCS '01*, pages 136–145, Washington, DC, USA, 2001. IEEE Computer Society. Full version available at http://eprint.iacr.org/2000/067.

Dam00. Ivan Damgård. Efficient Concurrent Zero-Knowledge in the Auxiliary String Model. In *Proceedings, Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 418–430, Berlin, Heidelberg, 2000. Springer.

DJN10. Ivan Damgård, Mads Jurik, and Jesper Buus Nielsen. A Generalization of Paillier's Public-key System with Applications to Electronic Voting. *International Journal of Information Security*, 9(6):371–385, 2010.

DN03. Ivan Damgård and Jesper Buus Nielsen. Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption. In *Proceedings, Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, Berlin,Heidelberg, 2003.

FPS00. Pierre-Alain Fouque, Guillaume Poupard, and Jacques Stern. Sharing Decryption in the Context of Voting or Lotteries. In *Proceedings, Financial Cryptography — FC 2000*, volume 1962 of *Lecture Notes in Computer Science*, pages 90–104, Berlin, Heidelberg, 2000. Springer.

GMR85. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The Knowledge Complexity of Interactive Proof-Systems. In *Proceedings, 17th Annual ACM Symposium on Theory of Computing — STOC '85*, pages 291–304, New York, NY, USA, 1985. ACM.

Gol06. Oded Goldreich. *Foundations of Cryptography: Volume 1, Basic Techniques.* Cambridge University Press, New York, NY, USA, 2006.

Gol09. Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

HP14. Carmit Hazay and Arpita Patra. One-Sided Adaptively Secure Two-Party Computation. In *Proceedings, Theory of Cryptography — TCC 2014*, volume 8349 of *Lecture Notes in Computer Science*, pages 368–393, Berlin, Heidelberg, 2014. Springer.

LP01. Anna Lysyanskaya and Chris Peikert. Adaptive Security in the Threshold Setting: From Cryptosystems to Signature Schemes. In *Proceedings, Advances in Cryptology — ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 331–350, Berlin, Heidelberg, 2001. Springer.

Nar. Isheeta Nargis. Efficient Oblivious Transfer for One-Sided Active Adaptive Adversaries. Accepted for publication in AFRICACRYPT 2014.

Nar17. Isheeta Nargis. Efficient Oblivious Transfer from Lossy Threshold Homomorphic Encryption. In *Proceedings, Progress in Cryptology — AFRICACRYPT 2017*, volume 10239 of *Lecture Notes in Computer Science*, pages 165–183, Cham, 2017. Springer.

NH24. Isheeta Nargis and Anwar Hasan. Covert Adaptive Adversary Model: A New Adversary Model for Multiparty Computation. Cryptology ePrint Archive, Paper 2024/729, 2024. https://eprint.iacr.org/2024/729.

NME13. Isheeta Nargis, Payman Mohassel, and Wayne Eberly. Efficient Multiparty Computation for Arithmetic Circuits against a Covert Majority. In *Proceedings, Progress in Cryptology — AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 260–278, Berlin, Heidelberg, 2013. Springer.

Pai99. Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proceedings, Advances in Cryptology — EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238, Berlin, Heidelberg, 1999. Springer.

PVW08. Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A Framework for Efficient and Composable Oblivious Transfer. In *Proceedings, Advances in Cryptology — CRYPTO '08*, volume 5157 of *Lecture Notes in Computer Science*, pages 554–571, Berlin, Heidelberg, 2008. Springer.