Efficiently Making Secure Two-Party Computation Fair

Handan Kılınç
EPFL, Switzerland

Alptekin Küpçü
Koç University, Turkey

Abstract

Secure two-party computation cannot be fair in general against malicious adversaries, unless a trusted third party (TTP) is involved, or gradual-release type of costly protocols with super-constant rounds are employed. Existing optimistic fair two-party computation protocols with constant rounds are either too costly to arbitrate (e.g., the TTP may need to re-do almost the whole computation), or require the use of electronic payments. Furthermore, most of the existing solutions were proven secure and fair separately, which, we show, may lead to insecurity overall.

We propose a new framework for fair and secure two-party computation that can be applied on top of any secure two party computation protocol based on Yao's garbled circuits. We show that our fairness overhead is minimal, compared to all known existing work. Furthermore, our protocol is fair even in terms of the work performed by Alice and Bob. We also prove our protocol is fair and secure simultaneously, through one simulator, which guarantees that our fairness extensions do not leak any private information. Lastly, we ensure that the TTP never learns the inputs or outputs of the computation. Therefore even if the TTP becomes malicious and causes unfairness, the security of the underlying protocol is still preserved.

Keywords: two party computation, garbled circuit, Yao's protocol, fair computation, optimistic model

1 Introduction

In two-party computation (2PC), Alice and Bob intend to evaluate a shared function with their private inputs. The computation is called secure when the parties do not learn anything beyond what is revealed by the output of the computation. Yao [47] introduced the concept of secure two-party computation and gave an efficient protocol; but this protocol is not secure against malicious parties who try to learn extra information from the computation by *deviating* from the protocol. Many solutions [39, 46, 30, 37] are suggested to strengthen Yao's protocol against malicious adversaries.

When one considers malicious adversaries, fairness is an important problem. A fair computation should guarantee that Alice learns the output of the function *if and only if* Bob learns. This problem occurs since in the protocol one party learns the output earlier than the other party; therefore (s)he can abort the protocol after learning the output, before the other party learns it.

There are two main methods of achieving fairness in 2PC: using gradual release [41, 32, 42] or a trusted third party (TTP) [10, 35]. The ***gradual release*** protocols [19, 8, 4] let the parties gradually (bit by bit, or piece by piece) and verifiably reveal the result. Malicious party will have one bit (or piece) advantage if the honest party starts to reveal the result first. Yet, if the malicious party has more computational power, he can abort the protocol earlier and he can learn the result via brute force, while the honest party cannot. In this case, fairness is not achieved. Another drawback of this approach is that it is expensive since it requires many rounds of communication.

The TTP approach employs a third party that is trusted by both Alice and Bob. Of course, a simple solution would be to give the inputs to the TTP, who computes the outputs and distributes fairly. In terms of efficiency and feasibility though, the TTP should be used in the ***optimistic*** model [2], where he gets involved in the protocol *only* when there is a dispute between Alice and Bob. It is very important to give the TTP the minimum possible workload because otherwise the system will have a bottleneck [10]. Another important concern against the TTP is ***privacy***. In an optimistic solution, if there is no dispute, the TTP should not even know a computation took place, and even with a dispute, the TTP should never learn the

identities of Alice and Bob, or their inputs or outputs. **We achieve all these efficiency and privacy requirements on the TTP.**

Another problem regarding fairness in secure two-party computation is the **proof methodology**. In previous work [10, 32, 41, 42], fairness and security were proven *separately*. However, it is important to prove them together to ensure that the fairness solution *does not leak* any information beyond the original secure two-party computation requirement. Therefore, as in the security of the secure two-party computation, there should be ideal/real world simulation (see Section 3) that covers *both fairness and security*. In other words, **the fairness solution should also be simulatable**. Besides, **the simulator should learn the output only after it is guaranteed that both parties can learn the output**.

Our Contributions: The main achievement of this work is an efficient framework for making secure 2PC protocols fair, such that it guarantees fairness and security together, and can work on top of any secure two party computation protocols based on Yao's garbled circuits [47].

- We use a simple-to-understand ideal world definition to achieve fairness and security together, and **prove our protocol's security and fairness together** according to it, using a single simulator.
- We show that the classic way of **proving security and fairness separately is not necessarily secure** (see Section 7).
- Our framework employs a trusted third party (TTP) for fairness, in the optimistic model [2]. The **TTP's load is very light**: verification of sigma (honest-verifier zero knowledge) proofs [17] and decryption only. If there is no dispute, the TTP does not even know a computation took place, and even with a dispute, the TTP never learns the identities of Alice and Bob, or their inputs or outputs. So, a semi-honest TTP is enough in our construction to achieve fairness. Additionally, even if the TTP becomes malicious (e.g., colludes with one of the parties), **it does not violate the security of the underlying 2PC protocol**; only the fairness property of the protocol is contravened.
- Our framework is also fair about the work done by Alice and Bob, since both of them perform almost the same steps in the protocol.
- The principles for fairness in **our framework can be adopted by any secure two party computation that is based on Yao's garbled circuits [47] and secure against malicious adversaries**, thereby achieving fairness with little overhead.
- We compare our framework with related fair secure two-party computation work and show that we achieve better efficiency and security.

2 Related Works

Secure Two-Party Computation: Yao [47] firstly presented a secure and efficient protocol for two-party computation in the *semi-honest model*, where the adversary follows the protocol's rules but he cannot learn any additional information from his view of the protocol. Since this protocol is not secure against malicious behavior, new methods were suggested based on proving parties' honesty via zero-knowledge proofs [25], which makes the protocol inefficient. Another approach is the cut-and-choose modification [37] to Yao's protocol, where we need $O(s)$ circuits for 2^{-s} error where s is the security parameter. Furthermore, the number of rounds is not constant and depends on s. However, there are improved versions of this technique [39, 46, 36, 21], as well as non-interactive versions [1]. Other techniques [10, 30] require parties to efficiently prove that their behavior is honest using efficient sigma-proof-based zero-knowledge proofs of knowledge.

Another problem in Yao's protocol is *fairness*. The general solutions are based on gradual release timed commitments [41, 32, 43, 42] and trusted third party (TTP) [10, 35].

Gradual Release: Pinkas [41] presents a complex method, where the evaluation of the garbled circuit is executed on the *commitments* of the garbled values rather than the original garbled values. It uses cut-and-choose method to prevent malicious behavior of the constructor and *blind signatures* [13] for the verification of the evaluator's commitments. However, this protocol is expensive because of the gradual release timed commitments. Moreover, there is a *majority circuit* evaluation that can reveal the constructor's input values, as shown by Kiraz and Schoenmakers [32], who improve Pinkas's construction by removing the majority circuit computation and the blind signatures, and using OR-proofs instead [17]. Yet, the other inefficiency

problems remain. Similarly, Ruan et al. [42] use Jarecki and Shmatikov construction [30], and instead of the proof of the correct garbled circuit, they employ the cut-and-choose technique. Gradual release, which is used to solve the fairness problem, constitutes the weak part regarding the efficiency.

Optimistic Model: Cachin and Camenisch [10] present a fair two-party computation protocol in the optimistic model. The protocol consists of two intertwined verifiable secure function evaluations. In the case of an unfair situation, the honest party interacts with the TTP. Escrowed oblivious transfers and many proofs used cause inefficiency. Even worse, **the job of the TTP can be as bad as almost repeating the whole computation**. Lindell [35] constructs a framework that can be adopted by any two-party functionality with the property that either both parties receive the output, or one party receives the output while the other receives a digitally-signed check (i.e., monetary compensation). However, one may argue that one party obtaining the output and the other obtaining the money may not always be considered fair, since *we do not necessarily know how valuable the output would be before the evaluation.*

Security Definitions: Pinkas [41] protocol does not have proofs in the ideal-real world simulation paradigm. The importance of ideal-real world simulation is explained by Lindell and Pinkas [38]. The standard simulation definition [26] does not include fairness because of the impossibility result of fairness without honest majority [15]. Garay et al. [23] relax the notion of fairness and defines a "commit-prove-fair-open" functionality to simulate gradual release to prove security and fairness together. Then, Kiraz and Schoenmakers [32] and Ruan et al. [42] use this functionality in their security proof but the output indistinguishabilty of the ideal and real worlds is not satisfied in their proofs because fairness is proven separately, without simulation. Therefore, none of these protocols has a proof in ideal-real world simulation paradigm that shows the protocol is both secure and fair. Cachin and Camenisch [10] give an ideal-real world definition that includes fairness and security together for protocols that employ a TTP. However, interestingly, they do *not* prove their proposed protocol according to their definition. Lindell [35] defines a new ideal world definition that captures security and fairness as defined above (i.e., exchanging money in return is considered fair as well [3, 33]) and proves fairness and security according to this definition. In Section 7, we show that proving fairness and security *separately* do *not* necessarily yield to **fair and secure** protocols. The security and fairness definition we use follows the same intuition as that of Canetti [12].

Finally, there are very efficient constructions for specific applications [28, 6, 20], but we are interested in computing general functionalities efficiently. We achieve this goal, fairly, in the malicious setting.

3 Definitions and Preliminaries

Yao's Two-Party Computation Protocol: We informally review Yao's construction [47], which is secure in the presence of *semi-honest* adversaries. Such adversaries follow the instructions of the protocol, but try to learn more information. The main idea in Yao's protocol is to compute a circuit without revealing any information about the value of the wires, except the output wires.

The protocol starts by agreeing on a circuit that computes the desired functionality. One party, called the *constructor*, generates two keys for every wire except the output wires. One key represents the value 0, and the other represents the value 1. Next, the constructor prepares a table for each gate that includes four double-encryptions with the four possible input key pairs (i.e., representing $00, 01, 10, 11$). The encrypted value is another key that represents these two input keys' output (e.g., for an AND gate, if keys k_{0_a} and k_{1_b} representing 0 and 1 are used as inputs of Alice and Bob, respectively, the gate's output key k', which is encrypted under k_{0_a} and k_{1_b}, represents the value $0 = 0\ AND\ 1$.). Output gates contain double-encryptions of the actual output bits (no more keys are necessary, since the output will be learned anyway). All tables together are called the *garbled circuit*.

The other party is the *evaluator*. The constructor and the evaluator perform oblivious transfer, where the constructor is the sender and the evaluator is the receiver, who learns the keys that represent his own input bits. Afterward, the constructor sends his input keys to the evaluator. The evaluator evaluates garbled circuit by decrypting the garbled tables in topological order, and learns the output bits. The evaluator can decrypt one row of each gate's table, since he just knows one key for each wire. Since all he learns for the intermediary values are random keys and only the constructor knows which value these keys represent, the evaluator learns nothing more (other than what he can infer from the output). The evaluator finally sends

the output to the constructor, who also learns nothing more than the output, since the evaluator did not send any intermediary values and they used oblivious transfer for the evaluator's input keys.

Secure Two-Party Computation: There are important requirements for a secure computation (e.g., privacy, correctness, independence of inputs, guaranteed output delivery, fairness [38]). Security is formalized with the ideal/real simulation paradigm. For every real world adversary, there must exist an adversary in the ideal world such that the execution in the ideal and real worlds are indistinguishable (see e.g., [24]).

Alice and Bob are trying to compute a function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$. Alice has her private input x, and Bob has his private input y. When they compute the function $f(x,y) = (f_a(x,y), f_b(x,y))$, Alice should obtain the output $f_a(x,y)$ and Bob should obtain the output $f_b(x,y)$. For general computation, fairness does not hold. Consider a malicious adversary in Yao's protocol, acting as the evaluator. He may choose not to send the output to the constructor.[1]

Optimistic Fair Secure Two-Party Computation: To provide *fair* secure two-party computation, we adapt the secure two-party computation definition, and add a third player to the system: the trusted third party (TTP). At the end of the computation, we require that either both Alice and Bob get their output values, or neither of them gets anything useful. For an *optimistic* protocol, the TTP is involved *only* when there is a dispute about fairness between Alice and Bob.

Ideal World: It consists of the corrupted party C, the honest party H, and the universal trusted party U (*not the TTP*). The ideal protocol is as follows:

1. U receives input x or the message ABORT from C, and y from H. If the inputs are invalid or C sends the message ABORT, then U sends $\perp$ to both of the parties and halts.
2. Otherwise U computes $f(x,y) = (f_c(x,y), f_h(x,y))$. Then he sends $f_c(x,y)$ to C and $f_h(x,y)$ to H.

The outputs of the parties in an ideal execution between the honest party H and an adversary $\mathcal{A}$ controlling C, where U computes f, is denoted $IDEAL_{f,\mathcal{A}(w)}(x,y,s)$ where x, y are the respective inputs of C and H, w is an auxiliary input of $\mathcal{A}$, and s is the security parameter.

The standard secure two-party ideal world definition [38, 26] lets the adversary $\mathcal{A}$ to ABORT *after* learning his output but *before* the honest party learns her output. Thus, proving protocols secure using the old definition would not meet the fairness requirements. Therefore, we prove our protocol's security and fairness under the definition above.

Real World: There is no universally trusted party U for a real protocol π to compute the functionality f. There is an adversary $\mathcal{A}$ that controls one of the parties, and there is the TTP T, who is involved in the protocol when there is unfair behavior. The pair of outputs of the honest party and the adversary $\mathcal{A}$ in the real execution of the protocol π, possibly employing the TTP T, is denoted $REAL_{\pi,T,\mathcal{A}(w)}(x,y,s)$, where x, y, w and s are like above.

Note that U and T are *not* related to each other. T is the part of the *real* protocol to solve the fairness problem when it is necessary, but U is not real (just an *ideal* entity).

Definition 1 (**Fair Secure Two-Party Computation**). *Let π be a probabilistic polynomial time (PPT) protocol and let f be a PPT two-party functionality. We say that π computes f* ***fairly and securely*** *if for every non-uniform PPT real world adversary $\mathcal{A}$ attacking π, there exists a non-uniform PPT ideal world adversary S so that for every $x, y, w \in \{0,1\}^*$, the ideal and real world outputs are computationally indistinguishable:*

$$\{IDEAL_{f,S(w)}(x,y,s)\}_{s\in\mathbb{N}} \equiv_c \{REAL_{\pi,T,\mathcal{A}(w)}(x,y,s)\}_{s\in\mathbb{N}}$$

For optimistic protocols, to simulate the complete view of the adversary, **the simulator also needs to simulate the behavior of the TTP** for the adversary. This simulation also needs to be indistinguishable.

Note that in our ideal world, the moment the adversary sends his input, U computes the outputs and performs fair distribution. Thus, the adversary can either abort the protocol before any party learns anything useful, or cannot prevent fairness. This is represented in our proof with a **simulator who learns the output only when it is *guaranteed* that both parties can learn the output**. Also observe that,

[1] Fairness against semi-honest adversaries is trivial, since they follow the protocol.

under this ideal world definition, **if the simulator learns the output and the adversary later aborts, that simulation would be *distinguishable*** from the ideal world.

Suppose that Alice is malicious and S simulates the behavior of honest Bob in the real world and the behavior of malicious Alice in the ideal world. Assume S learns the output of Alice from U in order to simulate the real protocol *before* it is guaranteed that in a real protocol both of the parties would receive their outputs. Further suppose that the adversarial Alice then aborts the protocol so that S does not receive his output in the real world. Thus, in the real world, S outputs $\perp$ as Bob's output, whereas the ideal Bob outputs the result of the computation. Clearly, the ideal and real worlds are *distinguishable* in this case. The proofs in [42, 32, 10] unfortunately fall into this pitfall.

Verifiable Escrow: An escrow is a ciphertext under the public key of the TTP T. A *verifiable* escrow [2, 11] enables the recipient to verify, using only the public key of T, that the plaintext satisfies some relation. A public non-malleable label can be attached to a verifiable escrow [45].

Communication Model: When there is dispute between the two parties, the TTP resolves the conflict *atomically*. We assume that the adversary cannot prevent the honest party from reaching the TTP before the specified time interval. We do not assume anything else about the communication model.

Notation: The garbled circuit that is generated by Alice is GC^A and the one generated by Bob is GC^B. We use A and B as a superscript of any notation to show to which garbled circuit they belong to. For simplicity of the presentation, assume Alice and Bob have l-bit inputs and outputs each. Alice's input bits are $x = \{x_1, x_2, ..., x_l\}$ and Bob's input bits are $\{y = y_1, y_2, ..., y_l\}$.

When we say Alice's input wires, it means that Alice provides the input for these wires. Similarly, Alice's output wires are those that correspond to the result of the computation learned by Alice. Bob's input and output wires have the matching meaning.

We have three kinds of gates: *Normal Gate*, *Input Gate*, and *Output Gate*. An *Input Gate* is a gate that has an input wire of Alice or Bob. Similarly, an *Output Gate* is a gate that has a wire of Alice's or Bob's output. A *Normal Gate* is a gate that is neither *Input Gate* nor *Output Gate*.

We use sets to group wires and gates. The notation G_I denotes the set of all *Input Gates*, and similarly G_O is for the set of *Output Gates*. The set W_O includes the *output wires* of all *Output Gates* and the set W_I includes the *input wires* of all *Input Gates*. The set of all wires is denoted with W.

Consider some notation $k^A_{w_b}$ in the protocol. The superscript A means that it belongs to the garbled circuit GC^A. w shows to which wire set it belongs. If $w \notin W_O \cup W_I$, b shows that it is the *right-side* wire. Instead of b, if we use a, it shows it is the *left-side* wire. For other wire sets, b shows that it is Bob's input or output wire. If it is a, it shows Alice's input or output wire. Similarly, W_{I_A} and W_{O_A} is the set of input and output wires of Alice, respectively. W_{I_B} and W_{O_B} are Bob's input and output wire sets.

Any small letter notation that have 0 or 1 in its subscript shows that it represents the bit 0 or 1. The capital letters are used for notation of the commitments, and if they have 0 or 1 in their subscript, it shows that they are for some value that represents 0 or 1. E_k shows an encryption with the key k. Therefore, $E_{k_1}E_{k_2}(m_1, m_2)$ means that m_1 and m_2 are both encrypted by the two keys k_1 and k_2.

4 Intuition of Our Fairness Solution

Remember that Yao's protocol is secure against *semi-honest* adversaries. When one considers *malicious* adversaries, care must be taken against the possible actions of the evaluator (E) and the constructor (C):

1. C can construct a circuit that does *not* evaluate the desired function. For example, he can construct a circuit computing $f(x, y) = y$, where x is his input and y is E's input, violating E's privacy.
2. C can use one *wrong* key and one *correct* key in the oblivious transfer phase where E learns his input keys from C. If E aborts, then C infers that E's input bit corresponded to the wrong key.
3. E can learn the output without sending it to C, causing *unfairness*.

The first two problems are about malicious behavior, and the last one is about fairness. Solutions to the first two problems are not our contribution; we employ previous works. Our fairness framework can be applied on top of any existing secure two-party computation solution that is secure against malicious adversaries and based on Yao's construction. The first problem can be solved via *efficient zero-knowledge*

proofs [30, 10] or *cut-and-choose technique* [46, 36, 41, 37], and the second problem can be solved via *committed oblivious transfer* (COT) [30, 31].

Failed Approaches: In the beginning it looks like adding *fairness* in a secure two party computation protocol with TTP does not need a lot of work. However, we should be careful because it can break the *security* of the protocol. Consider a very simple solution regarding constructor C, evaluator E, and the TTP. Assume that C constructs the circuit such that the output is not revealed directly, but instead the output of the circuit is an encrypted version of the real output, and C knows the key. Thus, after evaluation, E will learn this encrypted output, and C and E need to perform a fair exchange of this encrypted output and the key. However, there are multiple problems regarding this idea:

1. C can add wrong encryptions for the outputs of E, and hence E can learn wrong output result. The wrong encryptions can leak information about E's output value to C according to the abortion of E.
2. When there is dispute between E and C, and E goes to TTP for resolution, she cannot efficiently prove to the TTP that she evaluated the C's garbled circuit correctly.

In the scenario above, C has some advantages. Therefore, to remove C's advantage, E needs to be a constructor too. Thus, the idea is to make C and E construct their own circuits and then evaluate the other's circuit [40]. The following problems occur in the two-constructor case:

1. One of the parties can construct a circuit for a different functionality and so the other party may learn a wrong output.
2. Even when both E and C construct the same circuit, they can cause the other one to learn a wrong output by using different inputs for the two circuits.
3. If C and E can check their inputs are the same for both circuits, according to the two outputs from two circuits being equal or not, the malicious party can infer more information.

Besides the problems above, both C and E have to prove that they acted correctly during the protocol to learn output results from the TTP when there is dispute. These proofs should be efficient to reduce the workload on the TTP. In addition, the TTP should not learn anything about the inputs and outputs of the computation, as well as the identities of the participants. **Our solution achieves efficiency of and privacy against the TTP**.

Our Solution: We show how to efficiently add fairness property to any garbled-circuit-based secure two party computation protocol Γ using our framework. The four key points are: (i) Alice and Bob both act as the constructor and the evaluator, with almost equal responsibilities. There will be two garbled circuits: one that Alice constructs and Bob evaluates, and another that Bob constructs and Alice evaluates. (ii) These garbled circuits are constructed in same manner as in the Γ protocol, except the input and output gates. Alice and Bob make sure that they both construct the same garbled circuit by the technique (cut-and-choose or zero knowledge proofs) on the underlying protocol Γ. (iii) In addition, they show that they use the same inputs for both circuits by employing an *equality test* for inputs (see Appendix B). (iv) They also should obtain only some hidden form of the output bits, and must perform a fair exchange with the other party to obtain the plain output bits. Briefly, Γ becomes fair as follows:

1. [**Agreement to the Common Knowledge**] Alice and Bob learn the public key of the TTP, and Alice sends her verification key for the signature scheme to Bob. Alice and Bob jointly generate four random elements that are called *equality test constants*. They are designated for the left wire being 0 or 1 and the right wire being 0 or 1. These equality test constants will be used as bases (of exponentiation) in commitments, and will later be employed in checking for equality of inputs.
2. [**Construction of the Garbled Circuit**] Alice and Bob agree on a circuit that evaluates the desired functionality. Then, they individually randomly choose *input gate numbers* (as exponents) for each input gate to generate two *randomized equality test numbers* based on equality test constants for each wire of each input gate. According to their base, randomized equality test numbers represent 0 or 1. Then Alice commits to Bob's input wires' randomized equality test numbers in her circuit, and Bob commits to Alice's input wires' randomized equality test numbers in his circuit. In addition, they individually choose *random row pairs* for the rows of the garbled *output* gates and commit to them.

Then, they individually construct their garbled circuits as in the Γ protocol. The construction of the input and output gates differ, though. Each row of the garbled *input* table's encryptions has two encrypted values. One of them is the gate's output key and the other one is the decommitment of the randomized equality test numbers of the corresponding evaluator's wires. In the construction of garbled *output* tables, they encrypt random row pairs instead of the output bits.
When Alice and Bob each learn two randomized equality test numbers for each wire of each *input* gate after the evaluation of the circuits (see below), they can check if these are generated by the same equality test constants to check if both parties used their same input in both circuits (see Appendix B). In essence, we want to ensure that Alice used the same x and Bob used the same y in both circuits. In addition, the input gate numbers are private for the owner of the circuit because if one of the parties learns the other party's input gate numbers, it means s(he) learns the other party's input by using randomized equality test numberthat s(he) learned from evaluation. Therefore, the equality test is executed without revealing the input gate numbers using efficient zero knowledge proofs.
The random row pairs can be thought as unique identifiers for the rows of the constructor's garbled *output* gates. Only the constructor knows which row they represent, which means only the constructor knows which output bit they correspond to. This makes sure that the evaluator cannot learn the output directly, but must perform a fair exchange with the constructor.

3. [**Send, Prove and Evaluate**] Alice and Bob learn the keys associated with their input bits, in the garbled circuit the *other* party constructed, as in Γ. Alice sends all the necessary values to Bob for the evaluation of Alice's garbled circuit: all garbled tables and the keys of Alice's input bits as in Γ, together with the randomized equality test numbers of Alice's input. Similarly Bob sends to Alice the corresponding values for his garbled circuit. Then Alice and Bob each proves that they acted honestly up to this point by showing correctness of the Γ construction and fairness-related commitments. These proofs are necessary, since otherwise one party aborting due to a mis-constructed gate may let the constructor learn information about the other party's private input. If all proofs are valid, Alice sends her signatures on the commitment of random row pairs (Alice's signatures are necessary for resolutions with the TTP) and then they evaluate the garbled circuits and learn random row pairs of the other party's output wires. Otherwise, they abort the protocol.
4. [**Equality Test**] They execute the equality test in Appendix B to learn if they used the keys that represent the same bit for both circuits. For this purpose, they use the randomized equality test numbers that they learned during evaluation. At the end of evaluation, each party has a pair (representing the right and left wires) of randomized equality test numbers, which represent the same bits as the keys used, for each input gate. Therefore, if Alice and Bob used the same input in *both* garbled circuits, then the equality test succeeds. If the test is not successful, then they abort. Otherwise they continue and Alice signs a message that shows equality test is successful, and sends it to Bob. Note that the equality test is zero-knowledge, which means that neither party can gain any advantage, regardless of the test result.
5. [**Verifiable Escrow**] Alice sends a verifiable escrow to Bob that is encrypted with the TTP's public key. The verifiable escrow includes random row pairs of Bob's output gates that Alice learned from evaluation of Bob's garbled circuit. The verifiable escrow proof shows that Alice escrowed the correct output of Bob, since she learns correct random row pairs only after correct evaluation of the circuit. If there is problem in verification, then Bob aborts.
6. [**Output Exchange**] Firstly, Bob sends the random row pairs of Alice's output gates to Alice, because Bob already has a verifiable escrow and can resolve with the TTP should Alice choose not to respond. Alice checks if the random row pairs that Bob sent are correct. If they are not correct, she executes *Alice Resolve* with the TTP. Otherwise, Alice sends random row pairs of Bob's output gates to Bob. Then, Bob checks if the random row pairs that Alice sent are correct. If Alice does not respond or random row pairs are not correct, Bob executes *Bob Resolve* before the timeout. If something wrong goes on after the verifiable escrow phase, resolutions with the TTP guarantee fairness.

5 Making Secure 2PC Fair (Full Protocol)

Alice and Bob will evaluate a function $f(x,y) = (f_a(x,y), f_b(x,y))$, where Alice has input x and gets output $f_a(x,y)$, and Bob has input y and gets output $f_b(x,y)$, and for simplicity of the presentation we have $f : \{0,1\}^l \times \{0,1\}^l \to \{0,1\}^l \times \{0,1\}^l$, where l is a positive integer.

All commitments are Fujisaki-Okamoto commitments [22, 18] unless specified otherwise. and all (verifiable) encryptions are simplified Camenish-Shoup (sCS) [30] encryptions for our fairness framework. We give a review of the random numbers that are used for fairness in Table 1. The protocol steps are described in detail below.

Note that Appendix C has the adaptation of secure two party computation protocol [30] to the framework explained below.

Name	Form	Relation
Equality Test Constants	$e = g^\rho$	There are four kinds of them, where each represents 0 or 1 and right or left.
Input Gate Randoms	u	Each input gate has them. They are private; just known by the constructors.
Randomized Equality Test Numbers	$m = e^u$	For each input gate random u, there are four kinds of them, where each represents 0 or 1 and right or left according to e.
Random Row Pair	(δ, ε)	They are randomly chosen pairs each representing a row of the garbled output gates.

Table 1: The review of the random numbers used for fairness in our framework.

1. [**Agreement to the Common Knowledge**] The TTP generates a safe RSA modulus n and chooses a random element $g_s = (g')^{2n}$, where $g' \in \mathbb{Z}^*_{n^2}$ (for sCS encryption scheme). Then, the TTP selects random high-order elements $g, h \in_R \mathbb{Z}^*_n$ (for commitments). Additionally, he chooses a cyclic group $\mathcal{G}$ whose order is a large prime q and randomly selects its generators g_0, g_1, g_2 (for the equality test). Then, he announces his public key $PK_{TTP} = [(n, g_s, g, h), (\mathcal{G}, q, g_0, g_1, g_2)]$.
   Alice generates private-public key pair (sk_A, vk_A) for the signature scheme. She shares the public signature verification key vk_A with Bob.
   Lastly, they jointly generate four equality test constants $e_{a,0}, e_{a,1}, e_{b,0}$ and $e_{b,1}$. For joint generation, Bob first commits to four random elements from $\mathbb{Z}_n$ and proves knowledge of the committed values and their ranges [9, 5]. Then, Alice chooses four random elements from the same range and sends them to Bob. Finally, Bob decommits. Then they both calculate the multiplication of pairs of these numbers, in $\mathbb{Z}_n$, and so jointly agree on random numbers $\rho_{a,0}, \rho_{a,1}, \rho_{b,0}, \rho_{b,1}$. In the end, they calculate equality test constants where $\{e_{z,t} = g^{\rho_{z,t}} \ mod \ n\}_{z\in\{a,b\},t\in\{0,1\}}$. These numbers represent 0 and 1 for the left (a) and right (b) wires of the gates in G_I (input gates).
2. [**Construction of the Garbled Circuit**] Alice and Bob agree on the circuit that computes $f(x,y)$. Then, they begin to construct their garbled circuits separately. The construction is quite similar for Alice and Bob. Therefore, we give details of the construction through Alice. When there is an important difference, we also show Bob's way of doing it.
   Alice generates keys $\{k^A_{w_z,t}\}_{z\in\{a,b\},t\in\{0,1\},w\in W\setminus W_O}$ representing left and right and 0 and 1 for each wire except the output gates' output wires as in the protocol Γ. Then she constructs her garbled circuit GC^A as in the protocol Γ. She changes only the construction of *Output Gates* and *Input Gates* as explained below. There is no change in the construction of *Normal* gates. Additionally, she generates extra commitments for fairness-related random numbers.
   - **Commitments for fairness:** Alice chooses randomly one input gate number $u^A_g \in \mathbb{Z}_n$ for each input gate $g \in G_I$. Then she calculates the randomized equality test numbers m by raising equality test constants e to these random input gate numbers u as $\{m^{A_g}_{w^g_z,t} =$

$e_{z,t}^{u_g^A} \ mod \ n\}_{g\in G_I, z\in\{a,b\}, t\in\{0,1\}}$, where w^g is the wire of gate g. Here, each m value represents left or right wire and values 0 or 1 according to equality test constants used as a base. She prepares two commitments to the input gate number u_g^A for each right wire w_b^g (which corresponds to Bob's side) as $\{D^A_{w_b^g,t} = e_{b,t}^{u_g^A} h^{s_{w_b^g,t}} = m^A_{w_b^g,t} h^{s_{w_b^g,t}}\}_{g\in G_I, t\in\{0,1\}}$, where $s \in \mathbb{Z}_n$ is a random value picked to hide the committed value.

Bob does the same for GC^B, except he commits to the left (Alice's side) wires as $\{D^B_{w_a^g,t} = e_{a,t}^{u_g^B} h^{s_{w_a^g,t}} = m^B_{w_a^g,t} h^{s_{w_a^g,t}}\}_{g\in G_I, t\in\{0,1\}}$.

Remark: There are randomized equality test numbers, which represent 0 and 1 for the right and left wires, for each input gate, but **only the evaluator's side ones are committed** since the evaluator needs to learn one randomized equality test number that corresponds to his input. These numbers are required later to check whether or not the same input is used for both circuits in the evaluation. Hence, the name is "equality test numbers".

Note that there can be just one input wire of a gate (e.g., NOT gate for negation). In this case there will be two (instead of four) randomized equality test numbers which represent 0 and 1 for this gate. Alternatively, they can agree to construct a circuit using only NAND gates.

- **Input Gates:** The difference in the construction of a *Normal Gate* and an *Input Gate* is the following: Alice uses sCS encryption [30] to construct the input garbled table independent from the encryption scheme that Γ uses. She doubly-encrypts the partial decommitment of $D^A_{w_b^g,t}$, which is $s_{w_b^g,t}$, together with the output key, for each row, in a corresponding manner with using sCS encryption. In Bob's construction, he also doubly-encrypts the partial decommitment of $D^B_{w_a^g,t}$, which is $s_{w_a^g,t}$, together with the output key, for each row.

  Remark: Alice and Bob just encrypt partial decommitments because they only need to learn randomized equality test numbers (m values) that represents their input bits and we do *not* want to reveal input gate numbers (u values) since it may cause the evaluator to learn equality test constants that correspond to the constructor's input wires representing constructor's input bits.
- **Output Gates:** The garbled tables of *output gates* are constructed differently as well. Alice chooses random row pairs $(\delta_i^A, \varepsilon_i^A)$, each from $\mathbb{Z}_n$, for each row of all output garbled tables, where $i \in \{1, ..., 8l\}$. Then, she commits to the random row pairs that corresponds her output gates as $S_1^A, ..., S_{4l}^A$ (four commitments per output gate, assuming binary gates). Finally, she can construct the garbled output gates as a normal gate where the difference is that, instead of encryption of a key or the output bit, there is encryption of random row pairs. Similarly, Bob encrypts the random row pairs and prepares commitments $S_1^B, ..., S_{4l}^B$ for his output gates. These commitments will be used to prove correct construction of the output gates.

Remark: If the protocol Γ is based on cut-and-choose technique, in this case parties commits the same δ_i for each circuit using different ε_i^A. It is required because in the end of evaluation, the parties should choose the majority result of the all evaluated circuits. If δ_is would be different from each other, the parties could not learn the majority outputs of the circuits.

See Table 2 for construction details of output and input gates.

Row	Garbled Input Gate	Garbled Output Gate
00	$E_{k^A_{w_a,0}} E_{k^A_{w_b,0}} (s_{w_b^g,0}, k^A_{w_{o,0}})$	$E_{k^A_{w_a,0}} E_{k^A_{w_b,0}} (\delta_i^A, \varepsilon_i^A)$
01	$E_{k^A_{w_a,0}} E_{k^A_{w_b,1}} (s_{w_b^g,1}, k^A_{w_{o,1}})$	$E_{k^A_{w_a,0}} E_{k^A_{w_b,1}} (\delta_{i+1}^A, \varepsilon_{i+1}^A)$
10	$E_{k^A_{w_a,1}} E_{k^A_{w_b,0}} (s_{w_b^g,0}, k^A_{w_{o,1}})$	$E_{k^A_{w_a,1}} E_{k^A_{w_b,0}} (\delta_{i+3}^A, \varepsilon_{i+2}^A)$
11	$E_{k^A_{w_a,1}} E_{k^A_{w_b,1}} (s_{w_b^g,1}, k^A_{w_{o,1}})$	$E_{k^A_{w_a,1}} E_{k^A_{w_b,1}} (\delta_{i+3}^A, \varepsilon_{i+3}^A)$

Table 2: Garbled Input and Output Gates for an OR gate constructed by Alice. Input Gates employ sCS encryption, whereas the encryption scheme of Γ is used for the Output Gates.

3. [**Send, Prove, and Evaluate**] In this phase, Alice and Bob send all the necessary information to evaluate the garbled circuits as in Γ. Alice learns the her input keys that corresponds his input x from Bob as in Γ. Similarly Bob learns his input keys from Alice. They further perform the following:
Sending Fairness Values: Alice sends the input gate numbercommitments $\{D^A_{w_b^g,t}\}_{w_b \in W_{I_B}, g \in G_I, t \in \{0,1\}}$, the randomized equality test numbers $\{m^A_{w_a^g,x_w}\}_{g \in G_I}$ of her input wires, and the commitments $\{S_i^A\}_{i \in \{1,\ldots,4l\}}$ of random row pairs. Bob sends the commitments of input gate number $\{D^B_{w_a^g,t}\}_{w_a \in W_{I_A}, g \in G_I, t \in \{0,1\}}$, his input's randomized equality test numbers $\{m^B_{w_a^g,y_w}\}_{g \in G_I}$, and the commitments $\{S_i^B\}_{i \in \{1,\ldots,4l\}}$ of random row pairs.
After exchanging all garbled values, Alice and Bob prove to each other that they performed the construction honestly, by running these subprotocols (see Appendix A):
   - *Proof of Garbled Construction* to prove that the garbled tables are constructed properly as in Γ.
   - *Proof of Input Gates* to prove that the garbled input gates contain the correct decommitment values.
   - *Proof of Output Gates* to prove that the garbled output gates encrypt the committed random row pairs.

   If there is a problem in the proofs, they abort. Otherwise, Alice sends the signatures of the commitments of her random row pairs $\{sign_{sk_A}(S_i^A)\}_{i \in \{1,\ldots,4l\}}$.
   Then, they each evaluate the garbled circuit they were given. At the end of the evaluation, they just learn $2l$ random row pairs; l of them are for Alice's output gates and the other l of them are for Bob's output gates.
   Remark: If Γ is based on the cut-and-choose technique, Alice sends the signatures *only* for the *evaluation* circuits, not for the *check* circuits (see [37]) that are opened to check the correctness of the construction.
   Up to now, the main idea was to prove somehow (via zero-knowledge or cut-and-choose) that the garbled circuits were prepared honestly. Apart from our fairness additions, the underlying protocol remains the same. The evaluation is performed as usual. For example, if Γ uses cut-and-choose techniques, the parties evaluate more than one circuit and output the majority of the computation result of all the circuits. Since the committed δ values are the same for each circuit, they output the majority of the decommitments.
   Note that the constructor does not need to prove the randomized equality test numbers of *her own input wires*, since she has to send the correct ones to be successful in the equality test.

4. [**Equality Test**] *This part is necessary to test whether or not Alice and Bob used the same input bits for both circuit evaluations.* We use the idea by Boudot et al. [6]. There are fair and unfair versions of the equality test in [6], but the unfair version is sufficient for our purpose.
   Alice and Bob want to check, for the input gates in G_I, if $t'_b = t_b$ and $t'_a = t_a$ for the encryptions $E_{k^A_{w_a,t_a}}(E_{k^A_{w_b,t_b}}(k^A))$ and $E_{k^B_{w_a,t'_a}}(E_{k^B_{w_b,t'_b}}(k^B))$, such that the first one was decrypted by Bob and the second one was decrypted by Alice. For this purpose, we will use the randomized equality test numbers $\{m^A_{w_z^g,t}, m^B_{w_z^g,t}\}_{g \in G_I, z \in \{a,b\}, t \in \{0,1\}}$.
   Assume Alice decrypted a row for an input gate and learned two randomized equality test numbers $m^B_{g_a}, m^B_{g_b}$ for the input gate g. Also suppose Bob decrypted the same garbled gate and similarly learned $m^A_{g_a}, m^A_{g_b}$. If both used the same input bits for both GC^B and GC^A, then we expect to see that the following equation is satisfied:
$$(m^B_{g_a} m^B_{g_b})^{u^A_g} = (m^A_{g_a} m^A_{g_b})^{u^B_g} \tag{1}$$
   The left hand side of the equation (1) is composed of values Alice knows since she learned $m^B_{g_a}, m^B_{g_b}$ values from the evaluation of GC^B and generated u^A_g in the phase 2 of the protocol. Similarly, the right hand side values are known by Bob since he learned $m^A_{g_a}, m^A_{g_b}$ from the evaluation of GC^A and generated u^B_g in the phase 2 of the protocol. This equality should hold for correct values since $m^B_{g_a} = e^{u^B_g}_{w_a,t'_a}$, $m^B_g = e^{u^B_g}_{w_b,t'_b}$ and $m^A_{g_a} = e^{u^A_g}_{w_a,t_a}$, $m^A_{g_b} = e^{u^A_g}_{w_b,t_b}$.

After computing their side of these equalities for each input gate, they concatenate the results in order to hash them, where the output range of the hash function is $\mathbb{Z}_q$, where q is a large prime, which is the order of the prime-order group used for the subprotocol *Proof of Equality* in Appendix B. Then Alice and Bob execute this subprotocol with their calculated hash values as their inputs.
If the equality test succeeds, Alice signs the equality test result ($s_{eq} = sign_{sk_A}(equal)$), and sends it to Bob. If the signature verifies, they continue with the next step.
Remark: Remember that the constructor proved that all randomized equality test numbers in the encryptions are correct and each of them is a randomization of different equality test constants (see Appendix A.2). However, she did not prove that she added it to the correct row of the encryption table. Suppose that the constructor encrypted the randomized equality test number that represents 0 where the evaluator's encryption key represents 1. In this case, it is sure that the equality test will fail, but the important point is that the constructor *cannot* understand which row is decrypted by the evaluator, and thus does not learn any information because he cannot cheat just in one row. If he cheats in one row, he has to change one of the other rows as well, as otherwise he fails the "Proof of Input Gates". Thus, even if the equality test fails, the evaluator might have decrypted any one of the four possibilities for the gate, and thus might have used any input bit. This also means that the equality test can be simulated, and hence reveals nothing about the input.

5. [**Verifiable Escrow**] Alice creates a verifiable escrow by encrypting the random row pairs using the TTP's public key as $V = VE_{TTP}((\delta^B_{g^{i_1}_1}, \varepsilon^B_{g^{i_1}_1}), ..., (\delta^B_{g^{i_l}_l}, \varepsilon^B_{g^{i_l}_l}); vk_A, time)$. Here, g^i_j means that the j^{th} garbled output gate from G_O and i^{th} row of that gate. The part after the semicolon (;) is the public non-malleable label of the escrow. Then, she sends V to Bob. She proves that there are l different decommitments in the escrow that correspond to l of the commitments $S^B_1, ..., S^B_{4l}$ [29, 16, 7]. Since Alice can just decrypt one row for every gate, because she only has one pair of keys for each gate, this proof shows that Alice decrypted Bob's garbled output tables correctly, and the verifiable escrow has the information about which row was decrypted. If V fails to verify, or if the public signature verification key was different from what was used in the previous steps, or the *time* value is not what Bob was expecting (see [33]), then Bob aborts. Else, Bob continues with the next step.
Remark: We can replace the l out of $4l$ proof above with l proofs that are simply 1 out of 4 each (see [17]). To ensure each 1 out of 4 proof is for a different gate, Bob can use different commitment bases per gate. In this case, TTP's public key for the protocol extends to $O(l)$, and Alice proves, l times, for each output gate of Bob, that she knows one of the decommitments from the 4 commitments of the random row pairs for that gate.
If Γ is based on the cut-and-choose technique, then Alice chooses one of the circuits that gives the majority result, and adds the δ, ε pairs of this circuit to the verifiable escrow.

6. [**Output Exchange**] In this part, both Alice and Bob exchange the outputs. Remember that the outputs are indeed randomized, and only the constructor knows their meaning. Thus, if they do not perform this fair exchange, no party learns any information about the real output (unless they resolve with the TTP, in which case they both could learn their outputs). **At this point in the protocol, it is guaranteed that both parties can obtain the output**.
First, Bob sends the random row pairs that he learned for Alice's output gates in GC^A to Alice, since he already has a verifiable escrow as insurance. Alice checks if they are indeed the random row pairs that she generated. If they are, she learns the rows, equivalently the output bits, that correspond to her output. If Alice does not receive a correct response from Bob, then she runs *Alice Resolve*.
Then she sends Bob's random row pairs obtained from GC^B to Bob. Bob checks if the random row pairs are correct for his output wires. If they match his garbled construction, then Bob learns his output. If at least one of them is not matching or if Bob does not receive an answer from Alice (until some network timeout), he runs *Bob Resolve*.
Remark: Since both garbled circuits are generated for the same functionality and evaluated using the same inputs, the random row pairs reveal the output. Since Alice only learns her random row pairs in her circuit, she only learns her output and not Bob's. Similar argument applies to Bob's case.

5.1 Resolutions with the TTP

Bob Resolve: Bob has to contact the TTP before some well-specified timeout to get an answer [2, 33]. Considering he contacts before the timeout (verified using the *time* in the label of the verifiable escrow V), he first sends the verifiable escrow V of Alice, the equality signature s_{eq} to show that equality test was successful, and Alice's l random row pairs $(\delta_i^A, \varepsilon_i^A)$ that he learned from circuit evaluation along with the commitments and signatures of the commitments of these pairs $\{S_i^A, sign_{sk_A}(S_i^A)\}$ to prove that they are the correct representation of the decrypted row for the output values of Alice. The TTP checks if all signatures were signed by Alice, using the signature verification key vk_A in the verifiable escrow's label, and that the decommitments are all correct. If there is no problem, then the TTP decrypts the verifiable escrow V and sends the values inside to Bob. Since Bob knows which output wire he put these values in the garbled circuit he constructed, he effectively learns his output. The TTP remembers Alice's random row pair values, given and proven by Bob above, in his database.

Alice Resolve: When Alice contacts the TTP, she asks for her random row pairs. If the TTP has them, then he sends all random row pairs, which have already been verified in Bob Resolve, to Alice. If Bob Resolve has not been performed yet, then Alice should come after the timeout. When Alice connects after the timeout, if the TTP has the pairs, then he sends them to Alice. Otherwise she will not get any output, but she can rest assured that Bob also cannot learn any output values: The protocol is aborted. If Alice obtains random row pairs from the TTP, she can learn her real output results since these random row pairs uniquely define rows of the output gates of the garbled circuit she constructed.

Note that the **TTP learns nothing** about the protocol, since he only sees random row pairs, which are meaningless unless the corresponding bits are known, and a signature. Furthermore, the signature key can be specific to each circuit, and thus we do *not* require a public key infrastructure, and it does not give away Alice's identity. Hiding identities and handling timeouts in such resolution protocols are easy, as done by previous work [2, 33].[2]

6 Security of the Protocol

Theorem 1. *Let $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ be any probabilistic polynomial time (PPT) two-party functionality. The protocol above for computing f is secure and fair according to Definition 1, assuming that the subprotocols that are stated in the protocol are all secure (sound and zero-knowledge), all commitments are hiding and binding [22, 18], the signature scheme used is existentially unforgeable under adaptive chosen message attack [27] and the protocol Γ is a secure two party computation protocol based on Yao's garbled circuits.*

6.1 Proof Sketch

The important point in our proof is that after learning the input of the adversary, the simulator does *not* learn the output of the adversary from ideal world TTP U until it is guaranteed that both parties can obtain their outputs.

Malicious Alice: Simulator S_B prepares the circuit as the simulator of Γ with a random input y' and simulates all the proofs, including the equality test. S_B extracts the input x of Alice as in the simulation of Γ. Here, he does not send the input of Alice directly to the ideal world trusted party U. He waits until it is guaranteed that Bob can also obtain his input. If, before the verifiable escrow is received properly, the values that Alice sends are not correct or equality test is not successful, he sends message ABORT to U. After S_B receives the correct verifiable escrow, he sends Alice's input to U and receives Alice's output from U. At this point, he sends to Alice random row pairs of Alice's output by choosing the ones that match the output given by U. If he does not receive his correct output results from Alice, he does *Bob Resolve*.

Malicious Bob: Simulator S_A behaves almost the same as S_B, but now she contacts U once it is guaranteed that Alice would obtain her output as well. Since S_A does not know the actual output of Bob, she

[2]The discussion about the usage of unique identifiers for the TTP to identify different evaluations between some Alice and some Bob (with anonymity and unlinkability guarantees), as well as the timeout not harming the system, already exists in the optimistic fair exchange literature [2, 33], and therefore we are not complicating our presentation with such issues.

puts random values in the verifiable escrow and sends it to Bob, simulating the proof. Then S_A waits for the correct random row pairs of her output: if Bob sends them, then S_A sends to U Bob's input that he extracted and U sends back Bob's output. Then, S_A sends to Bob the correct random row pairs of Bob using the functionality output. However, if Bob does not send anything or sends incorrect random row pairs, she waits for Bob to perform Bob Resolve. If Bob does Bob Resolve, S_A acts as the TTP. If Bob performs a proper Bob Resolve, then S_A gives the input of Bob to U and learns the output of him. She then answers the resolution request accordingly. If Bob does not perform Bob Resolve until the timeout, S_A sends ABORT to U.

6.2 Full Proof

6.2.1 Security against Malicious Alice

Lemma 1. *Under the assumptions in Theorem 1, our protocol is secure and fair against malicious Alice.*

Proof. We construct an ideal-model adversary/simulator that interacts with Alice, the universal trusted third party U, and the fairness TTP T. The detailed behavior of the S_B follows:

1. [**Agreement to the Common Knowledge**] S_B generates the public key PK_{TTP} of the TTP and sends to $\mathcal{A}$, and $\mathcal{A}$ sends his signature verification key $vk_{\mathcal{A}}$ to S_B. Then, $\mathcal{A}$ and S_B decide on the equality test constants $\{e_{z,t}\}_{z\in\{a,b\},t\in\{0,1\}}$ that are generated jointly.
2. [**Construction of the Garbled Circuit**]
   - S_B and malicious Alice $\mathcal{A}$ agree on the circuit.
   - S_B picks random row pairs as in the protocol, and commits to them for every row of his garbled output gates. These commitments are formed correctly, as in the protocol.
   - S_B constructs the simulation of the garbled circuit same as the simulator of Γ. For the output gates, S_B adds an encryption of random row pair for every corresponding row of output garbled tables as in the protocol.
   - In the construction of the input garbled tables, S_B picks $\{\tilde{m}^B_{w^g_z,t}\}_{g\in G_I,z\in\{a,b\},t\in\{0,1\}}$ randomly instead of randomized equality test numbers.[3] Then he prepares commitments of $\{\tilde{m}^B_{w^g_a,t}\}_{g\in G_{I_a},t\in\{0,1\}}$ which are $\{\tilde{D}^B_{w^g_a,t}\}_{g\in G_{I_a}}$. Then he encrypts for the input garbled gates as in the protocol.
3. [**Send, Prove and Evaluate**]
   - $\mathcal{A}$ sends GC^A with all the necessary information including commitments and randomized equality test numbers m^A that represent her input. Similarly S_B sends the simulated garbled circuit $\tilde{GC}^B$ with all the necessary information along with fake commitments that he generated and $\tilde{m}^B$ values that represent his *random inputs*.
   - S_B learns the input of Alice $x = (x_1, ..., x_l)$ by using the simulator of Γ.
   - $\mathcal{A}$ performs all required proofs. S_B learns all random row pairs of $\mathcal{A}$'s output gates from the *Proof of Output Gates* and all input gate numbers $\{u_g\}_{g\in G_I}$ of GC^A from the *Proof of Input Gates*, using soundness extractors. If the adversary's proofs are not valid, then S_B sends message ABORT to U and stops. Otherwise, he starts to simulate the proofs he needs to make. Then, they continue with the evaluation of the garbled circuits.
4. [**Equality Test**] S_B acts as the random oracle, so when $\mathcal{A}$ asks for the hash of her equality test input, S_B answers her randomly and learns the equality test input of Alice, which consist of $\tilde{m}^B$ values. Since these were generated by S_B, he can check if they are the expected ones because S_B knows his own inputs and Alice's inputs for S_B's garbled circuit, so he knows which row $\mathcal{A}$ has to decrypt.
   In addition S_B checks if $\mathcal{A}$ used the same input as himself for the garbled circuit evaluation. He knows which row $\mathcal{A}$ should have decrypted, as explained above. He learned the input gate numbers $\{u_g\}_{g\in G_I}$

[3] Remember that in the real protocol, he exponentiates equality test constants with input gate numbers to compute randomized equality test numbers.

in the previous step. So S_B can learn which equality test constants of the row that he decrypted are used, and hence he can learn which input $\mathcal{A}$ is used in $\mathcal{A}$'s garbled circuit. Finally, S_B can check if the inputs that $\mathcal{A}$ used in both circuits are equal. If they are not equal then he sends ABORT message to U and the simulation ends.
Otherwise they begin the equality test protocol. If the proofs of $\mathcal{A}$ are not valid or the input values are not the expected ones, then S_B sends ABORT message to U. S_B simulates all zero-knowledge proof of knowledge protocols in the equality test, similarly as in [6]. If the test was supposed to return true, he simulates that way, otherwise he simulates with another random value whose hash is random. The probability that the adversary queries the random oracle at this random value is negligible.
If the equality test is successful, then $\mathcal{A}$ should send the signature s_{eq}. If S_B does not receive a verifying signature, he sends ABORT message to U.
Remark: Our security theorem intentionally does not mention the random oracle model. As is specified above and in [6], the equality test and the use of the hash function here necessitates the use of the random oracle model. Yet, one may convert this part to be secure in the standard model, with a loss of efficiency. First, all zero-knowledge proofs of knowledge must be done interactively. Second, instead of hashing the values and performing only a single equality test, the values must be compared independently, using $2l$ (sequential) equality tests. Performing these equality tests independently does not harm security, since one may not cheat in the other party's input, but can only cheat with own input, and therefore learns nothing from the (in)equality.

5. [**Verifiable Escrow**] $\mathcal{A}$ needs to send the verifiable escrow and if verification fails, then S_B sends ABORT message to U. Otherwise, the simulation continues with the next step.
6. [**Output Exchange**] At this point S_B is sure that both parties can obtain the output, because honest real Bob could have obtained his output using the Bob Resolve, and thus there is no more need for aborting. Hence, S_B hands Alice's input x to U and U sends $f_a(x, y) = \{f_{w,a}\}_{w \in W_{O,a}}$ to the simulator. Remember that he extracted all random row pairs of $\mathcal{A}$ from the *Proof of Output Gates*. In addition, he learned which bit the row represents from the *Proof of Garbled Construction*. Then, he picks random row pairs corresponding to the $\mathcal{A}$'s output $f_a(x, y)$ and sends all to $\mathcal{A}$.
S_B outputs whatever $\mathcal{A}$ outputs.

□

Claim 1. *The view of $\mathcal{A}$ in his interaction with the simulator S_B is indistinguishable from the view in his interaction with real Bob.*

Proof. We need to check the behaviors of the S_B that are different from Bob's behavior in the protocol, since these can affect the distribution.

- The construction garbled circuit of S_B is indistinguishable than the real one thanks to the simulator in protocol Γ.
- S_B did not commit to the real randomized equality test numbers as real Bob would do. Since the commitment scheme is hiding, the fake commitments are indistinguishable from the real ones. In addition, S_B simulates the zero-knowledge proofs in the protocol. Their indistinguishably comes from the zero-knowledge simulatability.
- The simulator only aborts when the real Bob would abort. Thus, the abort actions are indistinguishable as well.
- The simulator can perfectly simulate the TTP as well, since he generated its keys.

We emphasize one more time that a simulation proof for a secure and **fair two-party computation protocol needs to make sure that the simulator learns the output only when it is guaranteed that the other party can also obtain the output, and not before that point**. In real world, it is sure for honest Bob that he will learn the functionality output after Alice sends the correct verifiable escrow, because even if Alice does not send the output to him in the fair exchange phase, Bob can learn them from the TTP via *Bob Resolve*. For this reason S_B sends input x to U after receiving the valid verifiable escrow. After that Bob and S_B learn their outputs in the ideal world. As mentioned, Bob learns his output certainly

in the end of the protocol in the real world after this point. Since Bob is honest, $\mathcal{A}$ learns her output too, in the real world.

Overall, we proved that the joint distribution of $\mathcal{A}$ and the honest Bob's output in the real execution is indistinguishable from the joint distribution of S_B and the honest Bob's output in the ideal model. □

Finally, resolution protocols preserve fairness, as presented in the protocol description. Therefore, combined with Claim 1, the proof of Lemma 1 is complete.

6.2.2 Security against Malicious Bob

Lemma 2. *Under the assumptions in Theorem 1, our protocol is secure and fair against malicious Bob.*

Proof. The proof is similar to that of Lemma 1. We construct a simulator S_A that interacts as Alice with the adversary $\mathcal{B}$ that controls Bob in the real world, and as Bob with the U in the ideal world. S_A simulates the protocol as following:

1. [**Up to Verifiable Escrow**] All simulation actions are the same as the simulator S_B above until the end of *equality test*. There is just a minor difference, where initially S_A creates a key-pair for the signature scheme and sends her verification key vk_A to $\mathcal{B}$, and additionally sends the signature of the commitments of all the random row pairs to $\mathcal{B}$, as in the protocol. S_A sends the signature s_{eq} to $\mathcal{B}$ (if the equality test was indeed successful).
2. [**Verifiable Escrow**] S_A prepares the verifiable escrow. Since S_A does not know $f_a(x, y)$, she adds random values in the escrow. She labels the escrow with the verification key vk_A of the signature scheme and the correct *time* value. S_A simulates the proof by using verifiable escrow simulator [11].
3. [**Output Exchange**]
   - S_A waits for an answer in the output exchange phase. If $\mathcal{B}$ sends random row pairs, firstly S_A checks if they are random row pairs that she encrypted in the fake garbled circuit. If they are correct, we are guaranteed that the other party can obtain the output and the protocol will not be aborted. S_A sends y, which he learned via the Γ simulator, to U, and gets $f_b(x, y)$ from U. This also means that U sends $f_a(x, y)$ to Alice in the ideal world. Since S_A learned all decommitments of random row pairs of Bob from *Proof of Output Gates*, S_A prepares the random row pairs of $\mathcal{B}$ according to $f_b(x, y)$ and sends them to $\mathcal{B}$.
   - If there was a problem with the random row pairs that $\mathcal{B}$ sent, then resolution needs to take place. If before the timeout $\mathcal{B}$ runs *Bob Resolve*, S_A behaves like the TTP. First, S_A checks whether or not the verifiable escrow provided by $\mathcal{B}$ is the same as the one S_A sent in step 2, signature s_{eq} given by the adversary verifies under the public key of S_A in the verifiable escrow, and the random row pairs as proven by $\mathcal{B}$ are correct.
     If they are correct, then again it is guaranteed that both parties would obtain the output, and thus S_A sends y to U and gets $f_b(x, y)$ (and U sends $f_a(x, y)$ to ideal Alice). Then, S_A picks the correct random row pairs according to the output value $f_b(x, y)$. He sends these pairs to $\mathcal{B}$ as if they are the decryption of the verifiable escrow.
   - If $\mathcal{B}$ did not send output values to S_A in step 3, or did not properly perform *Bob Resolve* until the timeout, then S_A sends ABORT message to U.

□

Claim 2. *The view of $\mathcal{B}$ in his interaction with the simulator S_A is indistinguishable from the view in his interaction with real Alice.*

Proof. Honest Alice learns the output of the function if malicious Bob sends it in the output exchange phase, or if Bob successfully performs *Bob Resolve* in the real world. For this reason, the simulator S_A learns the output from U in the ideal world exactly at these points. The abort cases also follow the real world.

As the rest of the simulation is similar to the case in Claim 1, the joint distribution of $\mathcal{B}$ and honest Alice's output in the real execution is indistinguishable from the joint distribution of S_A and honest Alice's output in the ideal model. □

This completes the proof of Theorem 1.

7 Importance of Proving Security and Fairness Together

In this section we show the importance of proving security and fairness together according to Definition 1. We give a weird modified version of the protocol in Section 5. The modified protocol is almost same as the original. One change is in the "Sending Fairness Values" part of step 3. Here, Alice sends all signatures of the commitments of random row pairs $\{S_i^A, sign_{sk_A}(S_i^A)\}_{i\in\{1,\ldots,8l\}}$ instead of just sending signatures of commitments that correspond to her output wires.

The important changes are in the resolve protocols. In *New Bob Resolve*, Bob contacts the TTP before the timeout and gives all items as in the old Bob Resolve. In addition, he gives all signatures of $\{S_i^A\}_{i\in\{1,\ldots,8l\}}$ (in the original scheme he provided only $4l$ signatures) and l random row pairs that correspond to Alice's output along with l **random row pairs that correspond to his output**. In short, he gives information that can be used to identify both his and Alice's outputs as computed from Alice's circuit. Then, the TTP checks the correctness of the random row pairs by checking if they are decommitments of properly signed $\{S_i^A\}$s. In return, the TTP gives the decryption of the verifiable escrow as in the old Bob Resolve.

New Alice Resolve protocol's only difference from the old one is that the TTP gives Alice not only Alice's random row pairs but also Bob's random row pairs he obtained through the *New Bob Resolve*. Thus, Alice can learn both her and Bob's outputs. **These resolution protocols obviously violate privacy.**

We now show that this modified protocol **can be proven secure and fair separately**, via the standard method used in most of the previous work: prove security with the standard (unfair) simulation paradigm first, and then show fairness separately. Then, we show that such a protocol, and any of its privacy-violating variants, **cannot be proven secure and fair simultaneously** according to Definition 1.

Prove Security then Fairness: We first prove that the protocol is secure in the standard (unfair) ideal/real world definition. The security proof of this protocol is almost the same as our protocol's proof. The simulator learns the input of the adversary in the real world as the simulator of Γ and gives this input to U in the ideal world. Afterwards, the simulator receives output from U and continues the simulation as explained. If the adversary aborts the real protocol, then the simulator sends ABORT message to U.

For fairness of this protocol, note that either both parties receive the output or none of them receives, because if the protocol terminates before the verifiable escrow, no party can learn any useful information, and if the protocol finishes without problem, both of them learn the outputs. If the verifiable escrow is received, but the last phase of exchanging random row pairs was problematic, then the resolution protocols will ensure fairness such that both parties will learn their corresponding outputs if they resolve. Hence **according to this type of separate proofs, this contrived protocol is fair and secure**.

The problem arises since the fairness definition is only concerned that either Alice receives her output and Bob receives his, or no one receives anything. This definition of fairness is achieved by the contrived protocol. But, during resolutions, Alice learns some extra information. To prevent learning extra information during fairness extensions, one must also ensure that the simulation of the fairness parts are achievable. If the simulator is not concerned about fairness, one may prove such a contrived protocol fair, as we did above.

Try to Prove Security and Fairness: Let us try to prove security and fairness of such a contrived protocol using our joint definition. Assume, for simplicity, Alice is malicious. As in the proof of our original scheme, the simulator S_B learns the input x via Γ simulator, and then continues to simulate the view of the protocol using a random input y'. When the fairness is guaranteed, S_B sends Alice's input x to U, who responds back with $f_a(x, y)$.

After this point, S_B sends the corresponding random row pairs for $f_a(x, y)$ and waits for the answer from Alice. In the case that Alice does not answer or does not give the correct random row pairs, S_B performs *New Bob Resolve* with the TTP. Differently, from the original protocol, S_B has to give random row pairs that correspond to *both* $f_a(x, y)$ *and* $f_b(x, y)$. However, he *cannot* do it because he does not

know $f_b(x, y)$, or y, or anything that depends on y other than $f_a(x, y)$. The best that S_B can do is to give random row pairs that correspond to $f_b(x, y')$ and $f_a(x, y)$. After Alice comes for the *New Alice Resolve* and gets these random row pairs, she calculates the corresponding bits for each random row pair in her circuit and outputs $f_a(x, y), f_b(x, y')$. Unfortunately, this adversary's output is now **distinguishable** from the ideal world, since the ideal world of this protocol would correspond to outputting $f_a(x, y), f_b(x, y)$. We can conclude that this protocol is **not fair and secure according to our ideal/real world definition that captures security and fairness simultaneously**.

In general, using Definition 1, since the simulator S_B only knows $f_a(x, y)$ and nothing else that depends on the actual y, *no* fairness solution that leaks information on y (other than $f_a(x, y)$) can be simulated.[4]

Consequently, it is risky to prove fairness separate from the ideal/real world simulation. We do not claim that previous protocols [10, 32, 42] have security problems because of their proof techniques, but we want to emphasize that the old proof technique does *not* cover all security aspects of a protocol. Interestingly, Cachin and Camenisch [10] definition also prevents such problems, but they prove their protocol secure using the old style of proving security and fairness separately.

Importance of the Timing of the Simulator contacting the Universal Trusted Party: The proofs of the protocols [42, 32] are problematic since the simulator learns the output of the computation from U *before* it is guaranteed that the other party can also obtain the output. This behavior of the simulator **violates the indistinguishibility of the ideal and real worlds** because if the simulator does not receive his/her output in the real world while the parties already obtained the outputs in the ideal world, then the outputs in ideal and real worlds are distinguishable, and the simulation fails. Therefore, **the simulator must obtain the output from the universal trusted party in the ideal world, *only after* it is guaranteed that both parties can obtain the output.**

8 Comparison with Related Work

Our protocol efficiently achieves fairness and security in two-party computation against malicious adversaries. Table 3 compares our framework with previous work.

Advantages over Gradual Release based solutions: Fair 2PC solutions using gradual release [41, 32, 42] require much more number of rounds at the final output exchange stage, *proportional to the security parameter*. In comparison, our protocol adds extra 10 message sending steps for the common knowledge, output exchange, verifiable escrow, and the equality test (See Appendix B). In the case of a dispute, there is an extra round with the TTP. In comparison, an unfair scenario in the gradual release setting would possibly require extreme computational effort. In addition, when the parties' computational powers are not the same in such a setting, there can be unfairness.

Advantages over other TTP based solutions: The protocol by Camenisch and Cachin [10] does not use gradual release, but it uses the TTP inefficiently. In the resolution phase, a party has to send *the whole transcript* of the protocol to the TTP to show that (s)he behaved honestly. Then, the TTP and the party together run the protocol from where it stopped, which means possibly performing *almost the whole protocol* with the TTP.

Lindell's protocol [35] uses TTP as in our protocol, however its disadvantage is that one of the parties can get a coin instead of the output of the functionality. Even though this fairness definition presents itself in other contexts [3, 33], it is not suitable for two-party computation because no party knows the output of the functionality and so it is hard to estimate the value of the result beforehand.

In our protocol, **the workload of the TTP is very light**, since he only checks the correctness of the decommitments and signatures, and decrypts a verifiable escrow. The TTP does *not* learn inputs and outputs of the parties, and we do *not* require an external coin mechanism or putting monetary value on the output of the functionality. The TTP is unaware of any computation being performed unless there is a

[4] For symmetric functionalities where $f_a(x, y) = f_b(x, y)$ the simulation would be successful. But also notice that the contrived protocol is still fair in those cases, since learning the other party's output does not provide any additional information the party could not obtain via her/his own output.

dispute, and even when a dispute occurs, the parties remain **anonymous**, and their computations remain **unlinkable**. Therefore, a *semi-honest* TTP, who is employed only for fairness, is enough, and even if he becomes malicious and colludes with one participant, he **cannot violate the security of the protocol**.

Advantages of Proving Fairness and Security via Simultaneous Simulation: One of the most important contributions of our protocol is that it is proven secure and fair via simulating the protocol together with its fairness extensions and resolutions. In comparison, the security of the protocols in [41] are not proven with ideal/real world simulation. The protocols in [32, 42] are proven secure according to the standard ideal/real world paradigm definition, and then fairness is proven separately. The simulators in their proofs learn the output from the universal trusted party U of the ideal world *before* it is guaranteed that the other party would also obtain the output. Since we want that the ideal and real world distributions are indistinguishable, these proofs do not satisfy our definition, because it enables one of the parties (the simulator-adversary) abort after obtaining the output and before the other party obtains the output. Lindell's protocol [35] is proven to be fair and secure together, but for their own ideal/real world definition based on their protocol. Similarly, [10] has fair and secure ideal world definition, but their proof is done with the old style of proving security and fairness *separately*. In addition, U in ideal world has to contact the real world TTP T in every case; this harms the indistinguishably property of ideal and real worlds since maybe T never gets involved in the real protocol in an optimistic setting.

Advantages over fixed protocols: Existing work, except Lindell [35], propose a fixed secure 2PC protocol that is fair. Instead, we propose a *framework* that can be applied on top of any secure two party computation protocol that based on Yao's garbled circuits [30, 46, 36, 41, 37, 21]. This means, **we can enjoy the benefits of both cut-and-choose and zero-knowledge-proof based solutions**.

Lastly, it is possible that each party designs a circuit that calculates only her/his own output result. This can be done by just removing the output garbled gates of the evaluator, thereby simplifying the circuit and making all related techniques (proofs or cut-and-choose) slightly faster. Even further, Alice and Bob can construct and evaluate the garbled circuits in *parallel*, thus timewise we do *not* incur a real overhead against existing unfair solutions.

	[41]	[10]	[32]	[35]	[42]	Ours
Malicious Behavior	CC	ZK	CC	ZK/CC ✓	CC	ZK/CC ✓
Fairness	IGR	EOFE^i	IGR	EOFE^c	IGR	EOFE ✓
Proof Technique	NS	NFS	NFS	FS ✓	NFS	FS ✓

Table 3: Comparison of our protocol with previous works. CC denotes cut-and-choose, ZK denotes efficient zero-knowledge proofs of knowledge, IGR denotes inefficient gradual release, EOFE denotes efficient optimistic fair exchange, superscript i denotes *inefficient* TTP, superscript c denotes necessity of using *coins*, NS denotes *no* ideal-real simulation proof given, NFS indicates simulation proof given but *not for fairness*, and finally FS indicates *full simulation* proof including fairness. A ✓ is put for easily identifying better techniques.

Acknowledgements

The authors acknowledge the support of TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project number 111E019, and European Union COST Action IC1306.

References

[1] A. Afshar, P. Mohassel, B. Pinkas, and B. Riva. Non-interactive secure computation based on cut-and-choose. In *EUROCRYPT*, 2014.

[2] N. Asokan, V. Shoup, and M. Waidner. Optimistic fair exchange of digital signatures. *IEEE Journal on Selected Areas in Communications*, pages 591–610, 2000.

[3] M. Belenkiy, M. Chase, C. Erway, J. Jannotti, A. Küpçü, A. Lysyanskaya, and E. Rachlin. Making p2p accountable without losing privacy. In *ACM WPES*, 2007.

[4] D. Boneh and M. Naor. Timed commitments (extended abstract). In *CRYPTO*, 2000.

[5] F. Boudot. Efficient proofs that a committed number lies in an interval. In *EUROCRYPT*, 2000.

[6] F. Boudot, B. Schoenmakers, and J. Traoré. A fair and efficient solution to the socialist millionaires' problem. *Discrete Applied Mathematics*, 111(1-2):23–36, 2001.

[7] E. Bresson and J. Stern. Proofs of knowledge for non-monotone discrete-log formulae and applications. In *ISC*, 2002.

[8] E. F. Brickell, D. Chaum, I. Damgård, and J. v. d. Graaf. Gradual and verifiable release of a secret. In *CRYPTO*, 1987.

[9] Brownie cashlib cryptographic library. http://github.com/brownie/cashlib.

[10] C. Cachin and J. Camenisch. Optimistic fair secure computation. In *CRYPTO*, 2000.

[11] J. Camenisch and V. Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO*, 2003.

[12] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13:143–202, 2000.

[13] D. Chaum. Blind signatures for untraceable payments. In *CRYPTO*, 1982.

[14] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *CRYPTO*, 1993.

[15] R. Cleve. Limits on the security of coin flips when half the processors are faulty. In *STOC*, 1986.

[16] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO*, 1994.

[17] I. Damgård. On sigma protocols. http://www.daimi.au.dk/ ivan/Sigma.pdf.

[18] I. Damgard and E. Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In *ASIACRYPT*, 2002.

[19] I. B. Damgård. Practical and provably secure release of a secret and exchange of signatures. *Journal of Cryptology*, 8:201–222, 1995.

[20] C. Dong, L. Chen, J. Camenisch, and G. Russello. Fair private set intersection with a semi-trusted arbiter. In *DBSec*, 2013.

[21] T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In *EUROCRYPT*, 2013.

[22] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *CRYPTO*, 1997.

[23] J. Garay, P. MacKenzie, M. Prabhakaran, and K. Yang. Resource fairness and composability of cryptographic protocols. In *TCC*, 2006.

[24] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[25] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[26] S. Goldwasser and Y. Lindell. Secure computation without agreement. In *Distributed Computing*. Springer, 2002.

[27] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17:281–308, Apr. 1988.

[28] S. D. Gordon, C. Hazay, J. Katz, and Y. Lindell. Complete fairness in secure two-party computation. *Journal of ACM*, 58, 2011.

[29] R. Henry and I. Goldberg. Batch proofs of partial knowledge. In *ACNS*, 2013.

[30] S. Jarecki and V. Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, 2007.

[31] M. S. Kiraz and B. Schoenmakers. A protocol issue for the malicious case of yao's garbled circuit construction. In *In Proceedings of 27th Symposium on Information Theory in the Benelux*, 2006.

[32] M. S. Kiraz and B. Schoenmakers. An efficient protocol for fair secure two-party computation. In *CT-RSA*, 2008.

[33] A. Küpçü and A. Lysyanskaya. Usable optimistic fair exchange. *Computer Networks*, 56:50–63, 2012.

[34] A. Küpçü. *Efficient Cryptography for the Next Generation Secure Cloud: Protocols, Proofs, and Implementation*. Lambert Academic Publishing, 2010.

[35] A. Y. Lindell. Legally-enforceable fairness in secure two-party computation. In *CT-RSA*, 2008.

[36] Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO*, 2013.

[37] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, 2007.

[38] Y. Lindell and B. Pinkas. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality*, 1, 2009.

[39] J. B. Nielsen and C. Orlandi. Lego for two-party secure computation. In *TCC*, 2009.

[40] M. F. Payman Mohassel. Efficiency tradeoffs for malicious two-party computation. In *PKC*, 2006.

[41] B. Pinkas. Fair secure two-party computation. In *EUROCRYPT*, 2003.

[42] O. Ruan, J. Chen, J. Zhou, Y. Cui, and M. Zhang. An efficient fair uc-secure protocol for two-party computation. *Security and Communication Networks*, 2013.

[43] O. Ruan, J. Zhou, M. Zheng, and G. Cui. Efficient fair secure two-party computation. In *IEEE APSCC*, 2012.

[44] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, pages 161–174, 1991.

[45] V. Shoup and R. Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. In *EUROCRYPT*, 1998.

[46] D. P. Woodruff. Revisiting the efficiency of malicious two-party computation. In *EUROCRYPT*, 2007.

[47] A. C. Yao. Protocols for secure computations. In *FOCS*, 1982.

A Sub Protocols

We present the subprotocols from Alice's viewpoint as the prover. Bob's proofs are symmetric.

A.1 Proof of Output Gates

If the protocol Γ is based on the cut-and-choose technique, the verifier checks if the output gates have the correct decommitments, while checking the correctness of the circuit construction. If Γ is based on zero-knowledge proofs, the parties perform the following:

Inputs: Common inputs are the garbled tables of gate g that contains encryptions $E_{00}, E_{01}, E_{10}, E_{11}$, along with the commitments $S_i^A, ..., S_{i+3}^A$ of the random row pairs of g. Prover's private inputs are all decommitments. Prover performs following:

- $\boldsymbol{ZKVerifyEnc}(E_{00}, S_i^A) \wedge \boldsymbol{ZKVerifyEnc}(E_{01}, S_{i+1}^A) \wedge \boldsymbol{ZKVerifyEnc}(E_{10}, S_{i+2}^A) \wedge \boldsymbol{ZKVerifyEnc}(E_{11}, S_{i+3})$, where $\boldsymbol{ZKVerifyEnc}(E, S)$ proves that the encryption E contains the decommitment of S.

A.2 Proof of Correct Input Gates

Inputs: Common inputs are the commitments $D_{w_b,0}^A, D_{w_b,1}^A$ and the garbled input gate table that has encryptions $E_{00}, E_{01}, E_{10}, E_{11}$. Prover knows the decommitments. Prover performs the following (details in [30] or appendix of [34]):

- According to prover's side, he performs one of the proofs below. If she is from the left side (in our protocol it is Alice's side) then she performs $\boldsymbol{ZKVerifyEncRight}$, otherwise he performs $\boldsymbol{ZKVerifyEncLeft}$. Each $\boldsymbol{ZKVerifyEnc}(E, D)$ shows that the encryption E contains the decommitment of the commitment D. This proof can be done as in [11].
  - $\boldsymbol{ZKVerifyEncRight} = \boldsymbol{ZKVerifyEnc}(E_{00}, D_{w_b,0}^A) \wedge \boldsymbol{ZKVerifyEnc}(E_{01}, D_{w_b,1}^A) \wedge \boldsymbol{ZKVerifyEnc}(E_{10}, D_{w_b,0}^A) \wedge \boldsymbol{ZKVerifyEnc}(E_{11}, D_{w_b,1}^A)$
  - $\boldsymbol{ZKVerifyEncLeft} = \boldsymbol{ZKVerifyEnc}(E_{00}, D_{w_b,0}^A) \wedge \boldsymbol{ZKVerifyEnc}(E_{01}, D_{w_b,0}^A) \wedge \boldsymbol{ZKVerifyEnc}(E_{10}, D_{w_b,1}^A) \wedge \boldsymbol{ZKVerifyEnc}(E_{11}, D_{w_b,1}^A)$
- $\boldsymbol{ZKComDL}(D_{w_b,0}^A) \wedge \boldsymbol{ZKComDL}(D_{w_b,1}^A)$ proving knowledge of the decommitments of these commitments.
- $\boldsymbol{ZKEqComDL}$ $(D_{w_b,0}^A, D_{w_b,1}^A)$ showing that the committed messages are equal for both commitments (under different bases, $e_{b,0}$ and $e_{b,1}$).

B Equality Test

The adapted protocol of [6] is the following:

- Alice and Bob generate a cyclic abelian group G, which has a large prime order q, with generators g_0, g_1, g_2, g_3. Then, they compute the corresponding values of their input bits (for equality test) in $\mathbb{Z}_q$. In our case, these are the hashed values of all left (right) hand sides of equation 1. Say Alice's value is θ_a and Bob's is θ_b.
- Alice chooses a random $x_a \in \mathbb{Z}_q$ and computes $g_a = g_1^{x_a}$ and similarly Bob chooses a random $x_b \in \mathbb{Z}_q$ and computes $g_b = g_1^{x_b}$. Then, they send these values to each other and calculate $g_{ab} = g_a^{x_b} = g_b^{x_a}$. In addition, they prove knowledge of x_a and x_b using Schnorr's protocol [44].

- Alice selects $a \in_R \mathbb{Z}_q$, calculates $(P_a, Q_a) = (g_3^a, g_1^a g_2^{\theta_a})$, and sends (P_a, Q_a) to Bob. Bob does the symmetric version, computing and sending $(P_b, Q_b) = (g_3^b, g_1^b g_2^{\theta_b})$, where $b \in_R \mathbb{Z}_q$. Alice calculates $R_a = (Q_a/Q_b)^{x_a}$. Bob also calculates $R_b = (Q_a/Q_b)^{x_b}$. Then, Alice sends R_a with a proof that $log_{g_1} g_a = log_{Q_a/Q_b} R_a$ to Bob [14].
- Bob can now learn the result of the equality test by checking whether $P_a/P_b = R_a^{x_b}$. Then, he sends same proof as above to show $log_{g_1} g_b = log_{Q_a/Q_b} R_b$ so that Alice also learns the equality test result.

Note that if Bob does not send the last message, he will not obtain the equality signature s_{eq} in our protocol, and the whole protocol will be aborted without anyone learning the actual output.

C Making a Secure Protocol Fair

Alice and Bob will evaluate a function $f(x, y) = (f_a(x, y), f_b(x, y))$, where Alice has input x and gets output $f_a(x, y)$, and Bob has input y and gets output $f_b(x, y)$, and for simplicity of the presentation we have $f : \{0,1\}^l \times \{0,1\}^l \to \{0,1\}^l \times \{0,1\}^l$, where l is an integer.

All commitments are Fujisaki-Okamoto commitments [22, 18] unless specified otherwise, and all encryptions are simplified Camenish-Shoup (sCS) encryptions as in [30]. We use the protocol [30] to adapt our fairness principles in Section 5. We do not explain all the steps that are exactly the same as in Section 5.

1. [**Agreement to the Common Knowledge**]
2. [**Construction of the Garbled Circuit**] Alice and Bob agree on the circuit that computes $f(x, y)$. Then, they begin to construct their garbled circuits separately. The construction is quite similar for Alice and Bob. Therefore, we give details of the construction through Alice. When there is an important difference, we also show Bob's way of doing it.

   Commitments for keys: Alice generates keys $\{k^A_{w_z,t}\}_{z\in\{a,b\},t\in\{0,1\},w\in W\setminus W_O}$ representing left and right and 0 and 1 for each wire except the output gates' output wires. Then, she computes sCS commitments $\{C^A_{w_z,t}\}_{z\in\{a,b\},t\in\{0,1\},w\in W\setminus W_O}$ for each key as in [30].

   Commitments for fairness: It is same in Section 5.

   Normal Gates: As in the Yao's protocol [47], she begins to construct the garbled tables. First, she picks permutation pairwise bits φ_a^A and φ_b^A for each gate to permute the garbled table. Then she prepares the double encryptions of the keys according to permutation order (see Table 4).

   Input Gates and Output Gates: It is contructed same way as in Section 5 and the encryption is done with using simplified Camenish-Shoup (sCS) [30].
   See Table 4 for construction details of output and input gates.

Row	t_a, t_b	Output Bit (t)	Garbled Input Gate	Garbled Output Gate
00	$(0 \oplus \varphi_a^A), (0 \oplus \varphi_b^A)$	$(t_a \vee t_b)$	$E_{k^A_{w_a,t_a}} E_{k^A_{w_b,t_b}} (k^A_{w_o,t}, s_{w_b^g,t})$	$E_{k^A_{w_a,t_a}} E_{k^A_{w_b,t_b}} (\delta_i^A, \varepsilon_i^A)$
01	$(0 \oplus \varphi_a^A), (1 \oplus \varphi_b^A)$	$(t_a \vee t_b)$	$E_{k^A_{w_a,t_a}} E_{k^A_{w_b,t_b}} (k^A_{w_o,t}, s_{w_b^g,t})$	$E_{k^A_{w_a,t_a}} E_{k^A_{w_b,t_b}} (\delta_{i+1}^A, \varepsilon_{i+1}^A)$
10	$(1 \oplus \varphi_a^A), (0 \oplus \varphi_b^A)$	$(t_a \vee t_b)$	$E_{k^A_{w_a,t_a}} E_{k^A_{w_b,t_b}} (k^A_{w_o,t}, s_{w_b^g,t})$	$E_{k^A_{w_a,t_a}} E_{k^A_{w_b,t_b}} (\delta_{i+3}^A, \varepsilon_{i+2}^A)$
11	$(1 \oplus \varphi_a^A), (1 \oplus \varphi_b^A)$	$(t_a \vee t_b)$	$E_{k^A_{w_a,t_a}} E_{k^A_{w_b,t_b}} (k^A_{w_o,t}, s_{w_b^g,t})$	$E_{k^A_{w_a,t_a}} E_{k^A_{w_b,t_b}} (\delta_{i+3}^A, \varepsilon_{i+3}^A)$

Table 4: Garbled Input and Output Gates for an OR gate constructed by Alice.

3. [**Committed Oblivious Transfer (COT)**] There are two runs of the COT protocol: one where Alice is the sender and Bob is the receiver, and vice versa. They execute the following for each input bit of the receiver. In the case that Alice is the sender, Bob gives his input bit y_w to the functionality $\mathcal{F}_{\mathcal{COT}}$ [30] and Alice gives the keys $k_{w_b,0}, k_{w_b,1}$ for Bob's input wire to $\mathcal{F}_{\mathcal{COT}}$. Afterward, $\mathcal{F}_{\mathcal{COT}}$ gives the key k_{w_b,y_w} that represents his input bit y_w to Bob. The functionality ensures that the keys match the committed values. When the Bob is the sender, a similar process is executed.
4. [**Send, Prove, and Evaluate**] In this phase, Alice and Bob send all the necessary information to evaluate the garbled circuits as in Section 5 and prove those are built correctly.

- *Proof of Garbled Construction* to prove that garbled tables are constructed properly (as in [30]).
- *Proof of Correct Randoms* to prove garbled input gates contain correct decommitment values.
- *Proof of Output Gates* to prove that garbled output gates encrypt the committed random row pairs.

5. [**Equality Test**]
6. [**Verifiable Escrow**]
7. [**Output Exchange**]