Efficient Verification of Input Consistency in Server-Assisted Secure Function Evaluation

Vladimir Kolesnikov[1], Ranjit Kumaresan[2⋆], and Abdullatif Shikfa[3]

[1] Bell Labs, Murray Hill, NJ 07974, USA
kolesnikov@research.bell-labs.com
[2] University of Maryland, College Park, MD 20740, USA
ranjit@cs.umd.edu
[3] Bell Labs, 91620 Nozay, France
abdullatif.shikfa@alcatel-lucent.com

Abstract. We consider generic secure computation in the setting where a semi-honest server assists malicious clients in performing multiple secure two-party evaluations (SFE).
We present practical schemes secure in the above model. The main technical difficulty that we address is efficiently ensuring input consistency of the malicious players across multiple executions. That is, we show how any player can prove he is using the same input he had used in another execution. We discuss applications of our solution, such as online profile matching.

1 Introduction

Secure multiparty computation allows players to compute the value of a multivariate function on their inputs while keeping the inputs private. Generically, the problem of secure computation has been solved [39, 12, 7], for both semi-honest and malicious players. Since then, extensive body of work concentrated on optimizing these approaches so that they can be used in practice with acceptable overhead.

In this work, we consider a setting where the malicious players wish to securely compute on their inputs and are assisted by a semi-honest server to do so. This setting, although not fully general, is gaining prominence, due to the increasing success of collaborative platforms and online social networks. These platforms offer (and depend upon) a wide variety of applications that would benefit from privacy protection for their users. One such functionality is online profile matching (or match ratio computation). On dating sites, users search for other users whose characteristics/profile match their *desiderata*, while online job portals connect companies and job seekers who best match job openings. Typical applications considered for general SFE, such as auctions, payments, etc., are of interest also in our setting, with the added advantage of possible efficiency gain due to the opportunity to engage the help of the server.

⋆ Work partly done while the author was visiting Bell Labs. Work partly supported by NSF grant #1111599.

As mentioned, these and other scenarios naturally introduce a third-party, the server, who is in the position to assist the users with their computation, and who can also protect users' identities. Asking the server to perform the computation himself implies revealing private data, which is often sensitive and needs protection. At the same time, the server is an established business and often can be trusted not to deviate from the prescribed protocol. It is therefore reasonable to assume that the server is semi-honest.

In our approach, we will use Yao's Garbled Circuit (GC) [39] as the basic tool. See Section 3.1 for the description of the technique. Having access to the semi-honest server resolves the problem of malicious circuit generation. We also rely on Oblivious Transfer (OT) secure against malicious receiver and semi-honest sender. Combination of the two gives us a natural solution for our setting (see Section 2 for more details). This solution is complete for the standard standalone executions, usually considered in SFE research.

1.1 Input Consistency Verification

In this work we consider multiple SFE executions. One issue that arises here, and which is not addressed by the standard SFE model is that of input consistency between executions.

The need for input consistency. We first argue the importance of input consistency verification. We consider several motivating examples first.

Consider profile matching and match ratio computation. These are the underlying functionalities in online dating, resume/job matching, profiling for advertisement and other services, etc. In many of these applications, it is critical to the business model that users cannot manipulate their inputs to extract maximum benefit, but, rather, that user's inputs are consistent among executions.

Consider, for example, the online dating application, where Alice and Bob evaluate their compatibility by creating (and sometimes modifying) their profiles and matching them to their preferences. This process may be interactive, and the functions of interest may be adaptively selected. It may be important that the corresponding inputs provided in these functions, are chosen consistently. For example, if Bob's private profile indicates that he is working on Ph.D. in cryptography, and he later finds out that Alice likes kittens, he should not be able to later claim a veterinary or firefighting degree.

Similarly, if one system user, a corporation, is running a promotion campaign targeting a certain demographic, other users should not be able to improperly adjust their profiles to take advantage of the promotion.

High-level Approach. One natural approach to this issue is to have the server certify the players' profiles by issuing a certificate. This works well in limited circumstances, but not always. There are several problems with this approach. Firstly, certification is often understood as involving verification and approval. While certifying profile characteristics, such as *Age* is reasonable, certifying *favorite color* may look unusual. Further, some characteristics, such as *favorite color* are dynamic, and may change during the lifetime of the system. Finally, sometimes a need arises in considering a personal feature which was not

expected to be in the profile, as its usefulness may have been discovered during the interactive profile matching.

We consider a more light-weight approach, where no inputs are certified by anybody. However, once a certain input had been used by Alice in communication with Bob, Bob can always ask her to supply the same input in future communication, and vice versa. In our previous example, once Bob supplied the profile that indicated he studies cryptography, Alice will be able to ensure that in all future SFE where the field of studies is involved, Bob will input *cryptography*.

Similarly, if Alice communicated with corporation `CoffeeCorp`, and Alice's profile indicated she has graduated, she will not be able to participate in `CoffeeCorp`'s promotion for free coffee for college students.

We note that the fact that some inputs may change over the course of time will not break the fundamentals of our approach. The user with changed profile attribute might simply inform the other user, if needed, that a particular attribute was updated. The computation will go through, and other user will simply be additionally informed of the changed input.

1.2 Our Setting

We summarize our setting, which we motivated in the previous section.

Two parties, Alice and Bob want to evaluate a function, without disclosing their inputs to each other. Either of the parties can be maliciously corrupted. They are assisted by a semi-honest server $\mathcal{S}$ who does not collude with any of the other players. $\mathcal{S}$ has no input, and he obtains no output in the computation. We allow both Alice and Bob to verify, with the help of the server, that, for two SFE evaluations, a particular input wire is set to the same plaintext value.

We do not discuss which input wires are allowed to be checked, and how Alice and Bob agree on which wires they are checking. We assume that this agreement is reached over an insecure channel, and disagreement in this matter will simply result in non-participation in the SFE and/or input verification protocols.

In particular, importantly, we will not allow a player to verify consistency of two inputs of the other player without the other player's consent.

1.3 Related work

We consider SFE with malicious players, who use the help of a semi-honest server. Most relevant to us is a comparatively small body of work that provides improvements in settings similar to ours. We mention, but do not discuss in detail here the works that specifically concentrate on the malicious setting, such as [27, 21, 35, 29, 37, 34, 5, 8]. This is because malicious-secure protocols are much more costly than the protocols we are considering.

The issue of input consistency comes up in secure two-party computation. Cut-and-choose, a very popular technique, requires evaluation of a security-parameter number of circuits, and a consistency check among all the inputs of all the evaluated circuits [31, 27, 37, 29]. We stress that all these works solve a harder problem than ours, since we have the help of the semi-honest server.

This server-assisted computation model has been considered as early as [11], where the authors consider players A and B who wish to let a third party C learn the output of their computation. The helping oblivious server has often been appealed to in circumstances where such a player is natural, and where regular two- or multi-party SFE would have been too costly. One example is that of auctions [33], where secure computation is achieved with the help of two non-colluding servers. Here one (semi-honest) server creates garbled circuit implementing the auction, and the other (malicious) server evaluates it based on the inputs of the clients, which were submitted through a proxy Oblivious Transfer protocol. Several protocols have also been developed for the special case of secure auctions using only one server [17, 6, 9, 10]. More recently, [15] argues that the server model is well-suited for the web (where clients connect and interact only once with servers, and simultaneous availability of all clients is not possible) and present several protocols for a number of functions of interest. Other recent works [23, 30, 3] also consider secure computation in the server-assisted model but allow the server to be malicious. As mentioned earlier, protocols secure against malicious parties are much more expensive than the protocols we are considering. [22] similarly uses the helping server to overcome the non-simultaneous nature of survey submissions in survey processing. This work is incomparable to ours because, firstly, most of it concentrates on specific functions of interest (e.g., auctions). More importantly, our distinguishing feature is the input consistency verification across several executions, which is not considered in these works.

Input consistency checking is recognized in security literature as an important ingredient in system building. For example, [16] consider privacy-preserving data mining and identify the need for input consistency checking in computing specific functions of interest. They further propose to solve it by involving expensive public key techniques. This body of work is incomparable to ours, because, firstly, they consider specific problems, their solutions are often informal and presented without proofs (e.g. that of [16]), and further rely on expensive public-key tools.

Finally, input consistency can be achieved via Certificate Authority (CA) issuing credentials for players' inputs. However, as we discussed in Introduction, this approach is not sufficiently general, and is more costly than our proposed approach.

In contrast with all of the above approaches, we show how to ensure input consistency across several executions while only relying on a small number of symmetric-key primitives, and minimal additional storage by the players only (one bit per input to be cross-referenced), and no additional storage by the server other than one master secret of security-parameter length.

1.4 Our contributions and outline of the work

We propose a quite general solution to the reactive SFE among two malicious players and the helping non-colluding semi-honest server. Our main technical contribution is a technique to ensure input consistency among several executions.

Our solution is very efficient and is comparable to that of Yao's Garbled Circuit (GC) in the semi-honest model.

We start our presentation with a high-level description of our technical idea in Section 2. We then provide the overview of the preliminaries in Section 3, before presenting the detailed protocol and proof in Section 4. We discuss several natural extensions in Section 5.

2 Overview of our approach

We base our solution on Yao's GC [39] (and its state-of-the-art optimizations such as garbled row reduction [36], free-XOR [25]). GC is secure against malicious circuit evaluator and semi-honest circuit constructor, therefore we will have the semi-honest server $\mathcal{S}$ generate the garbled circuit for the chosen function (as communicated to $\mathcal{S}$ by both clients). As for inputs, we will use OT extension [20] secure against malicious receivers and semi-honest server [18]. Each player runs above OT with the server to obtain wire secrets corresponding to their input. Then they send these wire secrets to the other player (and receive the other player's input secrets). The computed GC is then sent by $\mathcal{S}$ to both players for evaluation (it is important to send the GC after the inputs have been delivered so that, e.g., players cannot abort based on the output of SFE). At this point, each player can complete GC evaluation and compute their output.

The above is a complete solution with the exception of the input consistency verification.

Our main contribution is precisely the method for input consistency verification. Our main idea is as follows. The input wire secrets in the constructed (by $\mathcal{S}$) garbled circuit will encode their corresponding plaintext values according to a secret stored by $\mathcal{S}$. This can be done, for example, by $\mathcal{S}$ choosing and storing a random bit b_i and setting the last bit of the 0-secret of wire i to b_i and the last bit of 1-secret to be $\neg b_i$. Now, when, say, Bob, receives Alice's wire secret from Alice, he will store the last bit of the wire of interest. Note that effectively the plaintext value of this wire is shared between $\mathcal{S}$ and Bob. Now, when Bob wishes to confirm that plaintext values of two of Alice's wires across two executions are the same, he simply needs to compare the XOR of the two values he stored with the XOR of the corresponding values stored by $\mathcal{S}$. If the XOR values are the same, then Alice supplied the same input. Indeed, in both good-behavior cases (Alice supplying either $0,0$ or $1,1$ in the two executions), Bob's stored bits will XOR to the XOR of the two stored bits of $\mathcal{S}$. We stress that this check can be done "in plaintext", i.e., simply by $\mathcal{S}$ sending the corresponding XOR value to Bob. This approach is symmetrically applied to both players.

Finally, we note that the server generates the encoding bits b_i using a PRFG, so he does not need to store any of the plaintext encoding bits, as he can always regenerate them from his master secret, client ids and SFE id. We note that including client ids into the circuit generation seed derivation is important. If not included, two malicious players P_1, P_2 might open an honest Alice's input of execution C_i by pretending that C_i was their prior execution.

We now make several observations regarding our presentation and the result.

Observation 1 *We stress that since we encode the wire-secret to wire-key correspondence in a single bit of the wire key, it is important that this correspondence can not be violated by a malicious player. For example, a semantically secure encryption may ignore the last bit of the key. If such an encryption were used in GC, a malicious player could flip the last bit and later falsely convince the verifier of input consistency. To prevent this, we ensure that players cannot malleate the received wire keys by having the server $\mathcal{S}$ send hashes of all wire keys to both players.*

Observation 2 *In our formal presentation, we will consider functions F that output the same value to both Alice and Bob. Our theorems and protocols can be naturally extended to cover the general case of the multi-output functionalities.*

Observation 3 *While we concentrate our discussion on* input *consistency, we can verify consistency of any wires of the evaluated circuits using a natural generalization of our approach.*

Observation 4 *The server will aid in consistency verification only if it is approved by both players by correspondingly conveying their consent to $\mathcal{S}$. We do not discuss how players know which wires are to be checked, we assume this is given to players as an additional and insecure input.*

Observation 5 *The server will aid in SFE only if the evaluated circuit is approved by both players by correspondingly sending the (identical to each other) circuit description $\mathcal{S}$. We do not discuss how players know which function they wish to compute; we assume this is given to players as an additional and insecure input.*

Observation 6 *Our protocols achieve* fairness *with respect to both clients. Recall, fairness requires that both participants of SFE learn the output simultaneously. In other words, players are not allowed to abort early after learning the output. Full, and even partial fairness is quite expensive to achieve (see e.g., [13, 14]) in the standard two-party setting, and full fairness comes "for free" in our setting.*

Observation 7 *Each player only needs to remember the bit corresponding to a single instance of a specific semantic input (say the first one) to achieve comparison. For example, Alice will need to remember only one instance of bits corresponding to Bob's age. Indeed, all future uses of a particular semantic input can be compared to its first use.*

3 Preliminaries and Notation

3.1 Garbled Circuits (GC)

Yao's Garbled Circuit approach [39], excellently presented in [28], is the most efficient method for one-round secure evaluation of a boolean circuit C. We

summarize its ideas in the following. The circuit *constructor* $\mathcal{S}$ creates a *garbled circuit* $\widetilde{C}$: for each wire w_i of the circuit, he randomly chooses two garblings $\widetilde{w}_i^0, \widetilde{w}_i^1$, where $\widetilde{w}_i^j$ is the *garbled value* of w_i's value j. (Note: $\widetilde{w}_i^j$ does not reveal j.) Further, for each gate G_i, $\mathcal{S}$ creates a *garbled table* $\widetilde{T}_i$ with the following property: given a set of garbled values of G_i's inputs, $\widetilde{T}_i$ allows to recover the garbled value of the corresponding G_i's output, but nothing else. $\mathcal{S}$ sends these garbled tables, called *garbled circuit* $\widetilde{C}$ to the *evaluator* $\mathcal{C}$. Additionally, $\mathcal{C}$ obliviously obtains the *garbled inputs* $\widetilde{w}_i$ corresponding to inputs of both parties: the garbled inputs $\widetilde{x}$ corresponding to the inputs x of $\mathcal{S}$ are sent directly and $\widetilde{y}$ are obtained with a parallel 1-out-of-2 oblivious transfer (OT) protocol [32, 2, 28]. Now, $\mathcal{C}$ can evaluate the garbled circuit $\widetilde{C}$ on the garbled inputs to obtain the *garbled outputs* by evaluating $\widetilde{C}$ gate by gate, using the garbled tables $\widetilde{T}_i$. Finally, $\mathcal{C}$ determines the plain values corresponding to the obtained garbled output values using an output translation table received from $\mathcal{S}$. Correctness of GC follows from the way garbled tables $\widetilde{T}_i$ are constructed.

We note that GC evaluator cannot deviate from the prescribed protocol, and GC is therefore secure against malicious GC evaluator, given an appropriate malicious-secure OT protocol.

3.2 Notation

Let κ be the computational security parameter. The server $\mathcal{S}$ assists parties P_1 and P_2 to secure evaluate arbitrary functions over their inputs multiple times. In each iteration, $\mathcal{S}$ will be provided circuit C_i that party P_1 with client id id_1, and input x_i, and party P_2 with client id id_2, and input y_i, wish to evaluate. We let $x_{i,j}$ (resp. $y_{i,j}$) denote the j-th bit of x_i (resp. y_i). We assume that x_i (resp. y_i) is of length m (resp. n), and that I_1 (resp. I_2) represents the set of P_1's (resp. P_2's) input wires in C_i.

Server $\mathcal{S}$ maintains a master secret – a state, denoted by σ, across executions. Given a circuit C_i, and state σ, the server uses algorithm $\mathsf{GarbGen}(i, C_i, \mathsf{id}_1, \mathsf{id}_2, \sigma)$ to generate a garbled version of C_i which we denote by $\widetilde{C}_i$.

In circuit C_i, we let $u_{i,j}$ (resp. $v_{i,j}$) denote the j-th input wire belonging to party P_1 (resp. P_2). For a wire $u_{i,j}$ (resp. $v_{i,j}$), we refer to the garbled values corresponding to 0 and 1 by $\widetilde{u}_{i,j}^0, \widetilde{u}_{i,j}^1$ (resp. $\widetilde{v}_{i,j}^0, \widetilde{v}_{i,j}^1$) respectively. While evaluating the garbled circuit, the evaluator will possess only one of two garbled values for each wire in the circuit. We let $\widetilde{w}'_{i,j}$ denote the garbled value on wire $w_{i,j}$ that is possessed by the evaluator.

Our protocols are designed in the random oracle model. In our protocols H, E, and H' represent hash functions that are modeled as non-programmable random oracles.

4 Input Consistency Verification in Server-Assisted SFE

We start this section with the definition of security. Then, in Section 4.2, we present a simple protocol for server-assisted SFE secure against malicious play-

ers. This is the natural protocol based on GC. Finally, in Section 4.3, we show how to allow to perform input consistency checks.

4.1 Definitions

We provide a formal definition of the SFE functionality we will realize. We want P_1 and P_2 to securely compute with the help of the server, and, in addition, to be able to check each other's inputs for consistency. Our functionality provides two interfaces, evaluate and check, which, respectively, evaluate the given circuit, and checks two inputs for consistency. It is easy to see that such a functionality provides the utility we desire, i.e., allows P_1 to ensure consistency of P_2's inputs, and vice versa.

Definition 1. *The* consistency checking functionality $\mathcal{F}_{\mathsf{cc}}$ *is a reactive functionality that interacts with a server* $\mathcal{S}$ *and parties* P_1 *and* P_2 *in the following way.*

- *If it receives an* evaluate *request from both parties, then* $\mathcal{F}_{\mathsf{cc}}$ *obtains* (i, C_i, x_i) *from* P_1*, and* (i, C_i, y_i) *from* P_2*. If the values* i, C_i*, provided by* P_1 *and* P_2 *differ, then* $\mathcal{F}_{\mathsf{cc}}$ *returns* $\perp$ *to* $\mathcal{S}$*.* $\mathcal{F}_{\mathsf{cc}}$ *records* (i, x_i, y_i)*. Then* $\mathcal{F}_{\mathsf{cc}}$ *returns* $(i, z_i = C_i(x_i, y_i))$ *to both* P_1 *and* P_2*. Finally,* $\mathcal{F}_{\mathsf{cc}}$ *returns* (i, C_i) *to* $\mathcal{S}$*.*
- *If it receives a* check *request from both parties, then* $\mathcal{F}_{\mathsf{cc}}$ *obtains the same string* (i_1, j_1, i_2, j_2) *from both parties.* $\mathcal{F}_{\mathsf{cc}}$ *returns* (i_1, j_1, i_2, j_2) *to* $\mathcal{S}$*. Let* $w_{i,j}$ *denote the plaintext value carried on* j*-th wire in circuit* C_i *when evaluated using inputs* x_i *and* y_i*.* $\mathcal{F}_{\mathsf{cc}}$ *checks if* $w_{i_1,j_1} = w_{i_2,j_2}$*, and returns* pass *to both parties if the check passed, else it returns* fail *to both parties.*

We say that a stateful protocol π is a server-assisted secure computation protocol that allows consistency checking across multiple sequential executions if it securely realizes $\mathcal{F}_{\mathsf{cc}}$ in the presence of an adversary $\mathcal{A}$.

Remark: Note that the above definition reveals each circuit C_i as well as each check request (i_1, j_1, i_2, j_2) to the semi-honest server. This is allowed in our model, and is consistent with the standard definitions of SFE security. However, even this information can be hidden, and we discuss natural ways to do so in Section 5.

4.2 Server-Assisted Secure Computation

In this section, we describe a simple protocol for secure computation that supports multiple executions in the server-assisted setting. Our protocol is a natural adjustment of the secure computation protocols based on garbling schemes [39, 4]. The main idea of the protocol of this section is that the semi-honest server will generate the GC and distribute it to both players, after running the OT protocol. We note that we do not yet address input consistency in this section, and the following protocol is simply a building block. We will use the protocol of this section as a subprotocol in the scheme presented in Section 4.3.

In this protocol, $\widetilde{C}_i$ used by $\mathcal{S}$ is constructed as described in Section 3.1. Let input keys for P_1 be $\{\widetilde{u}^0_{i,j}, \widetilde{u}^1_{i,j}\}_{j\in I_1}$, and those corresponding to P_2 be $\{\widetilde{v}^0_{i,j}, \widetilde{v}^1_{i,j}\}_{j\in I_2}$. We describe our protocol below.

Protocol 1.

1. $\mathcal{S}$ and P_1 participate in m OT instances in the following way. In the j-th instance:
   - $\mathcal{S}$ acts as sender with input $(\widetilde{u}^0_{i,j}, \widetilde{u}^1_{i,j})$.
   - P_1 acts as receiver with input $x_{i,j}$.
   - P_1 obtains $\widetilde{u}'_{i,j}$ as output.
2. $\mathcal{S}$ and P_2 participate in n OT instances in the following way. In the j-th instance:
   - $\mathcal{S}$ acts as sender with input $(\widetilde{v}^0_{i,j}, \widetilde{v}^1_{i,j})$.
   - P_2 acts as receiver with input $y_{i,j}$.
   - P_2 obtains $\widetilde{v}'_{i,j}$ as output.
3. For each $j \in I_2$, $\mathcal{S}$ sends $H(\widetilde{v}^0_{i,j}), H(\widetilde{v}^1_{i,j})$ in random order to P_1. Let P_1 receive these as $\{g_{i,j}, g'_{i,j}\}_{j\in I_2}$.
4. For each $j \in I_1$, $\mathcal{S}$ sends $H(\widetilde{u}^0_{i,j}), H(\widetilde{u}^1_{i,j})$ in random order to P_2. Let P_2 receive these as $\{h_{i,j}, h'_{i,j}\}_{j\in I_1}$.
5. P_1 sends $\{\widetilde{u}'_{i,j}\}_{j\in I_1}$ to P_2.
6. P_2 sends $\{\widetilde{v}'_{i,j}\}_{j\in I_2}$ to P_1.
7. P_1 aborts the protocol if for some $j \in I_2$, $H(\widetilde{v}'_{i,j}) \notin \{g_{i,j}, g'_{i,j}\}$ holds.
8. P_2 aborts the protocol if for some $j \in I_1$, $H(\widetilde{u}'_{i,j}) \notin \{h_{i,j}, h'_{i,j}\}$ holds.
9. $\mathcal{S}$ sends the garbled circuit $\widetilde{C}_i$ to both P_1 and P_2.
10. Using keys $\{\widetilde{v}'_{i,j}\}_{j\in I_2}$ and $\{\widetilde{u}'_{i,j}\}_{j\in I_1}$, P_1 and P_2 evaluate $\widetilde{C}_i$ to obtain output z_i.

In the above protocol, we assume that H is a random oracle, and the encryption scheme (used to create garbled tables) is semantically secure, and the OT protocol is secure against a malicious receiver. Given this, it is easy to see that Protocol 1 allows for multiple secure evaluations in the presence of an adversary that either passively corrupts $\mathcal{S}$, or actively corrupts one of P_1, P_2. (A formal proof is included in the proof of Theorem 8.)

We will use Protocol 1 as a subprotocol to construct our main protocol that also enables consistency verification in addition to secure computation.

4.3 Verifying Consistency Across Multiple Executions

Our main technical contribution is a design of a new garbling scheme that will allow efficient consistency verification in our setting. Recall that Yao's garbled circuit is constructed by choosing for each wire $w_{i,j}$, garblings $\widetilde{w}^0_{i,j}, \widetilde{w}^1_{i,j}$ at random from $\{0,1\}^\kappa$, and creating the garbled tables $\widetilde{T}_{i,j}$ using any semantically-secure encryption scheme.

GC Encryption. In our GC garbling schemes, we employ the following encryption. For simplicity of presentation, we work in the random oracle model.

Let $E : \{0,1\}^* \rightarrow \{0,1\}^\kappa$ be a random oracle. For encrypting the value x in the truth table of the ℓ-th gate in the i-th execution, we use the following encryption scheme that takes two keys k_a, k_b as follows:

$$\mathsf{Enc}_{k_a,k_b}(x,i,\ell) = E(k_a \| k_b \| i \| \ell) \oplus x$$

Before presenting our main protocol, we describe our main amendment to the traditional Yao GC-based garbling scheme (and their benefits) that we take advantage of.

Correlating the keys across multiple executions. We achieve verifiable consistency across multiple executions by correlating the keys at the input level. More precisely, the server $\mathcal{S}$ chooses at random his master secret $\sigma \in \{0,1\}^\kappa$, permanently stores it, and uses it in the following way to covertly "mark" each input wire with its plaintext label. Let P_1's (resp. P_2) id be id_1 (resp. id_2), and $H' : \{0,1\}^* \rightarrow \{0,1\}$ be a one-bit RO. For $w_{i,j}$ that is the input wire of either P_1 or P_2: (1) the first $\kappa - 1$ bits of $\widetilde{w}^0_{i,j}, \widetilde{w}^1_{i,j}$ are picked at random, and (2) the last bit of $\widetilde{w}^0_{i,j}$ is set to $H'(i\|j\|\mathsf{id}_1\|\mathsf{id}_2\|\sigma)$, while the last bit of $\widetilde{w}^1_{i,j}$ is set to $\widetilde{w}^0_{i,j}$'s complement $1 \oplus H'(i\|j\|\mathsf{id}_1\|\mathsf{id}_2\|\sigma)$. As we will formally show below, correlating the keys in the manner described above will allow for efficient consistency verification. We stress that the remaining keys (i.e., those that do not correspond to input or output wires of C_i) are still picked at random from $\{0,1\}^\kappa$.

We are now ready to formalize the above discussion and to describe $\mathsf{GarbGen}$ which is the algorithm $\mathcal{S}$ uses to create the garbled circuit for the i-th execution. The algorithm $\mathsf{GarbGen}$ takes the execution index i, the circuit C_i, and the server's state (master secret) σ to produce garbled circuit $\widetilde{C}_i$.

In $\mathsf{GarbGen}$, we generate the wires garblings at random. We note that in practice, we would probably use a PRFG such as AES.

Algorithm $\mathsf{GarbGen}(i, C_i, \mathsf{id}_1, \mathsf{id}_2, \sigma)$**.**
In C_i, let $u_{i,j}$ and $v_{i,j}$ represent the input wires corresponding to P_1 whose client id is id_1 and P_2 whose client id is id_2 respectively.

- For every $w_{i,j}$ that is an input wire of either P_1 or P_2, do the following:
  1. Set $\widehat{w}_{i,j} := H'(i\|j\|\mathsf{id}_1\|\mathsf{id}_2\|\sigma)$. (Recall, H''s output is one-bit.)
  2. Choose $r_0, r_1 \leftarrow \{0,1\}^{\kappa-1}$ at random.
  3. Set $\widetilde{w}^0_{ij} := r_0 \| \widehat{w}_{i,j}$.
  4. Set $\widetilde{w}^1_{ij} := r_1 \| (1 \oplus \widehat{w}_{i,j})$.
- For every internal wire $w_{i,j}$ of C_i, choose $\widetilde{w}^0_{i,j}, \widetilde{w}^1_{i,j} \leftarrow \{0,1\}^\kappa$.
- For each gate G in C_i do the following: Let the gate index of G be ℓ. Suppose w_1 and w_2 represent input wires, and w_3 represent the output wires of gate G. For $j \in \{1,2,3\}$, let $\widetilde{w}^0_j, \widetilde{w}^1_j$ represent the garblings corresponding to 0 and 1 respectively. Given this, the garbled table $\widetilde{T}$, corresponding to gate G with gate function g, in $\widetilde{C}_i$ consists of a random permutation of the set $\{E(\widetilde{w}^{b_1}_1 \| \widetilde{w}^{b_2}_2 \| i \| \ell) \oplus \widetilde{w}^{g(b_1,b_2)}_3\}_{b_1,b_2 \in \{0,1\}}$. (Recall E is a random oracle.)

We are now ready to describe our final protocol that enables efficient verification of consistency across multiple executions. We refer the reader to technical overview of the protocol and its intuition presented in Section 2.

Protocol 2

Setup: $\mathcal{S}$ chooses random seed $\sigma \leftarrow \{0,1\}^\kappa$.

Evaluation: In the i-th execution:

- P_1 and P_2 provide C_i to $\mathcal{S}$. If submissions of P_1 and P_2 differ, the server aborts.
- $\mathcal{S}$ creates $\widetilde{C}_i \leftarrow \mathsf{GarbGen}(i, C_i, \mathsf{id}_1, \mathsf{id}_2, \sigma)$.
- $\mathcal{S}$ uses $\widetilde{C}_i$ as the garbled circuit, and participates in Protocol 1 with P_1 and P_2. At the end of the protocol, P_1 and P_2 obtain their respective outputs of the execution.
- For $w_{i,j}$ that represents the input wire of either P_1 or P_2, both P_1 and P_2 do the following.
  - Set $\widehat{w}'_{i,j}$ to the last bit of $\widetilde{w}'_{i,j}$.
  - Add $\widehat{w}'_{i,j}$ to the local state.

Consistency Verification:

- Both P_1 and P_2 specify (i_1, j_1, i_2, j_2) to $\mathcal{S}$. If submissions of P_1 and P_2 differ, the server aborts.
- $\mathcal{S}$ retrieves $\widehat{w}_{i_1,j_1} \leftarrow H'(i_1 \| j_1 \| \mathsf{id}_1 \| \mathsf{id}_2 \| \sigma)$, and $\widehat{w}_{i_2,j_2} \leftarrow H'(i_2 \| j_2 \| \mathsf{id}_1 \| \mathsf{id}_2 \| \sigma)$. $\mathcal{S}$ sends the bit $(\widehat{w}_{i_1,j_1} \oplus \widehat{w}_{i_2,j_2})$ to both P_1 and P_2. Denote the bit received by P_1 and P_2 as c.
- P_1 and P_2 retrieve $\widehat{w}'_{i_1,j_1}$ and $\widehat{w}'_{i_2,j_2}$ from their local state, and check if $c \stackrel{?}{=} \widehat{w}'_{i_1,j_1} \oplus \widehat{w}'_{i_2,j_2}$. If the check fails then the execution is aborted.

This completes the description of our protocol. We stress that unlike P_1 and P_2, the server $\mathcal{S}$ does not store any local state other than the master secret σ. We now provide the proof of security.

Theorem 8. *Let $\mathcal{A}$ be an adversary that either passively corrupts $\mathcal{S}$ or actively corrupts one of P_1, P_2. Then, Protocol 2 securely realizes $\mathcal{F}_{\mathsf{cc}}$ in the presence of $\mathcal{A}$.*

Proof. (sketch) **Security against semi-honest $\mathcal{S}$.** We start with showing that the protocol is secure against the semi-honest server $\mathcal{S}$. The information received by $\mathcal{S}$ are transcripts of the underlying OTs. This does not leak information, because OTs are secure against semi-honest $\mathcal{S}$. Given this, the simulator $\mathsf{Sim}_{\mathcal{S}}$ follows naturally.

Secure against malicious P_1^*. We present the simulator Sim_1 of a malicious P_1^*, and argue that it produces a good simulation.

Sim_1 chooses random seed $\sigma \leftarrow \{0,1\}^\kappa$. Sim_1 does the following in each iteration i.

Sim_1 first obtains $\widetilde{C}_i$ by running $\mathsf{GarbGen}(i, C_i, \sigma)$. Then, Sim_1 starts P_1^* and interacts with it, sending it messages it expects to receive, and playing the role of the trusted party for the OT oracle calls that P_1^* makes, in which P_1^* plays the role of the receiver.

Sim_1 plays OT trusted party m times, where P_1^* is the receiver; as such, Sim_1 receives all m OT selection bits (which are supposed to correspond to P_1^*'s input) from P_1^* and each time, for concreteness say j, uses $\{\widetilde{u}_{i,j}^0, \widetilde{u}_{i,j}^1\}$ (which are P_1's input keys specified by $\mathsf{GarbGen}$) to hand to P_1^* as his OT output. Let us denote this output by $\widetilde{u}_{i,j}$. If any of the underlying OTs abort, then Sim_1 sends abort to the trusted party and halts, outputting whatever P_1^* outputs.

Acting as $\mathcal{S}$, the simulator Sim_1 chooses random κ-bit string r_j, and sends $r_j, H(\widetilde{v}_{i,j}^0)$ in random order. Then, acting as P_2, Sim_1 receives $\{\widetilde{u}'_{i,j}\}_{j\in I_1}$ from P_1^*, and sends $\{\widetilde{v}_{i,j}^0\}_{j\in I_2}$ to P_1^*. Next, Sim_1 checks if for every $j \in I_1$, it holds that $\widetilde{u}'_{i,j} = \widetilde{u}_{i,j}$. If any of the checks fail, then the simulator sends abort to the trusted party, and terminates outputting whatever P_1^* outputs. Otherwise, Sim_1 forms x_i^* in the following way. For each $j \in I_1$, if $\widetilde{u}_{i,j} = \widetilde{u}_{i,j}^0$, then $x_{i,j}^*$ is set to 0; else it is set to 1. Then, Sim_1 feeds x_i^* as its input to the trusted party. Sim_1 gets back the output z_i from the trusted party.

Sim_1 creates a "fake" garbled circuit $\widetilde{C}'_i$ which is exactly the same as an honestly generated garbled circuit $\widetilde{C}_i$ (created using $\mathsf{GarbGen}$) except with the following modification. Let G denote an output gate with input wires w_1, w_2, and output wire w_3. Let G's gate index be ℓ. Recall Sim_1 obtained output z_i from the trusted party, so it knows the actual value b that is carried on the output wire w_3. Sim_1 creates garbled gate $\widetilde{T}$ as a random permutation of the set $\{E(\widetilde{w}_1^{b_1} \| \widetilde{w}_2^{b_2} \| i \| \ell) \oplus \widetilde{w}_3^b\}_{b_1,b_2\in\{0,1\}}$. That is, the "fake" garbled circuit, upon evaluation, will always yield the correct output z_i. Sim_1 sends the fake garbled circuit $\widetilde{C}'_i$ to P_1^*. This completes the description of the simulation of the evaluation phase.

We now describe the simulation of the consistency verification stage. Acting as $\mathcal{S}$, the simulator Sim_1 receives query (i_1, j_1, i_2, j_2) from P_1^*. Sim_1 checks if this is a valid query, and if not, it aborts the protocol, and terminates the simulation outputting whatever P_1^* outputs. Otherwise, Sim_1 returns $H(i_1 \| j_1 \| \mathsf{id}_1 \| \mathsf{id}_2 \| \sigma) \oplus H(i_2 \| j_2 \| \mathsf{id}_1 \| \mathsf{id}_2 \| \sigma)$ to both parties.

We now argue that Sim_1 produces a view indistinguishable from the real execution in both the evaluation and verification stages. We first note that Sim_1's interaction with P_1^* is indistinguishable from that of honest $\mathcal{S}$ and P_2. Indeed, OT secrets delivered to P_1^* are distributed identically to real execution. Further, since non-selected OT secrets remain hidden, P_1^* knows the value of exactly one key in $\{\widetilde{u}_{i,j}^0, \widetilde{u}_{i,j}^1\}$ for each of his input wires $j \in I_1$. Thus, if the execution is not aborted, then P_1^* must have sent the key that he obtained via OT with Sim_1 (acting as $\mathcal{S}$). Also, the fake garbled circuit sent to P_1^* from Sim_1 is indistinguishable from real, since the underlying encryption scheme is based on RO and produces uniform and independent ciphertexts.

Now we argue that the simulation of the consistency verification stage is indistinguishable from the real execution. Let (i_1, j_1, i_2, j_2) represent the query

that was sent by P_1. First, we consider the case when wires j_1 and j_2 carry P_2's inputs. Since valid queries over (honest) P_2's inputs are actually consistent in the real execution, and since Sim_1 generates keys honestly according to $\mathsf{GarbGen}$, we conclude that Sim_1's answer is indistinguishable from $\mathcal{S}$'s answer in the real execution. Now suppose wires j_1 and j_2 carry P_1's inputs. Since an RO collision happens with negligible probability, we are guaranteed that the key $\widetilde{u}'_{i,j}$ that P_1 sent to P_2 is exactly the one that P_1 retrieved via OT. Indistinguishability of the simulation follows from the fact that Sim_1 answers the query honestly based on keys generated via $\mathsf{GarbGen}$. This completes the proof of correctness of the simulation when P_1^* is malicious.

The case when P_2 is malicious is symmetric and is skipped. ■

5 Extensions

As observed earlier, our definition of $\mathcal{F}_{\mathsf{cc}}$ functionality reveals information about the verification queries and the circuit C_i to the server $\mathcal{S}$. While, as we discussed, this is not a security violation, in some use cases, it is desired to hide this information as well. In this section, we discuss natural approaches to hiding this information.

Private verification queries. We provide a simple solution for preserving privacy of verification queries. Recall that in order to verify input consistency, parties need to retrieve the XOR of the least significant bits of the wire keys corresponding to wires specified in their queries. Note that these least significant bits can be obtained directly from the private state σ (i.e. the master secret) of $\mathcal{S}$. This motivates the following solution that preserves privacy of verification queries. Consider a circuit C' which takes as input queries q_1, q_2 from parties P_1 and P_2, and the private state σ of $\mathcal{S}$, and computes the desired output (i.e., XOR of the least significant bits of the keys specified by the queries). Clearly, if C' is evaluated securely, i.e., while keeping queries q_1, q_2 private from $\mathcal{S}$, and private state σ hidden from P_1 and P_2, then our problem is solved. We propose the following efficient solution to securely evaluate C' using garbled circuits.

We model H' as a PRF (as opposed to RO) in order to allow C' to internally generate the keys from σ. (We stress that modeling H' as a PRF does not violate the security of our construction in any way.) We also require C' to check if $q_1 = q_2$ holds, and produce output only when this check passes. This is necessary in order to guarantee that the malicious party does not obtain information other than what the output of the honest query reveals.

Now, without loss of generality, suppose party P_1 wishes to verify input consistency. Parties simply securely evaluate C' on corresponding inputs, and use the output of this computation as discussed in consistency verification subprotocol of Protocol 2 described in Section 4.3. Clearly, $\mathcal{S}$ will not know which wires are being verified. We note that two colluding players P_1 and P_2 will not obtain output related to a third player, since H' used for evaluation of the marker bits is evaluated on inputs which include both client ids.

We stress that the above solution is very efficient[4]; in particular its complexity is independent of the number of past executions between P_1 and P_2.

Function privacy. We employ standard techniques such as universal circuits [38, 1, 26, 24] to preserve function privacy. Our application is mildly complicated by the fact that we need to ensure that both parties provide the same function descriptors as input to the universal circuit. This is done to prevent a malicious party from evaluating an arbitrary function over the honest party's inputs. We resolve this issue using techniques similar to the ones we employed when privacy of queries needed to be preserved.

In more detail, let U' denote a circuit that takes as input two function descriptors f_1, f_2, and two inputs x and y. Circuit U' checks if $f_1 = f_2$, and if so, evaluates a universal circuit U on input f_1, x, y to produce output $f_1(x, y)$. Clearly, if U' is evaluated securely, i.e., while keeping function descriptors f_1, f_2 private from $\mathcal{S}$, then our problem is solved. Secure evaluation of U' is performed in the same way as described in the setting where privacy of queries needed to be preserved.

References

1. Abadi, M., and Feigenbaum, J. Secure circuit evaluation. *Journal of Cryptology 2*, 1 (1990), 1–12.
2. Aiello, W., Ishai, Y., and Reingold, O. Priced oblivious transfer: How to sell digital goods. In *Advances in Cryptology – EUROCRYPT 2001* (May 2001), B. Pfitzmann, Ed., vol. 2045 of *Lecture Notes in Computer Science*, Springer, pp. 119–135.
3. Asharov, G., Jain, A., Lopez-Alt, A., Tromer, E., Vaikuntanathan, V., and Wichs, D. Multiparty computation with low communication, computation and interaction via threshold fhe. In *Advances in Cryptology – EUROCRYPT 2012* (2012), Lecture Notes in Computer Science, Springer, pp. 483–501.
4. Bellare, M., Hoang, V. T., and Rogaway, P. Foundations of garbled circuits. p. *To Appear at ACM CCS 2012*.
5. Bendlin, R., Damgård, I., Orlandi, C., and Zakarias, S. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology – EUROCRYPT 2011* (May 2011), K. G. Paterson, Ed., vol. 6632 of *Lecture Notes in Computer Science*, Springer, pp. 169–188.
6. Cachin, C. Efficient private bidding and auctions with an oblivious third party. In *Proceedings of the 6th ACM conference on Computer and communications security* (New York, NY, USA, 1999), ACM, pp. 120–127.
7. Chaum, D., Crépeau, C., and Damgård, I. Multiparty unconditionally secure protocols. In *20th Annual ACM Symposium on Theory of Computing* (May 1988), ACM Press, pp. 11–19.
8. Damgard, I., Pastro, V., Smart, N., and Zakarias, S. Multi-party computation from somewhat homomorphic encryption. In *Advances in Cryptology – Crypto 2012*, pp. 643–662.

[4] See [19] for the concrete cost of securely evaluating AES.

9. Di Crescenzo, G. Private selective payment protocols. In *Financial Cryptography*, vol. 1962 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001, pp. 72–89.
10. Di Crescenzo, G. Privacy for the stock market. In *Financial Cryptography*, vol. 2339 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 269–288.
11. Feige, U., Kilian, J., and Naor, M. A minimal model for secure computation (extended abstract). In *STOC'94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing* (1994), ACM, pp. 554–563.
12. Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game, or a completeness theorem for protocols with honest majority. In *19th Annual ACM Symposium on Theory of Computing* (May 1987), A. Aho, Ed., ACM Press, pp. 218–229.
13. Gordon, S. D., Hazay, C., Katz, J., and Lindell, Y. Complete fairness in secure two-party computation. In *40th Annual ACM Symposium on Theory of Computing* (May 2008), R. E. Ladner and C. Dwork, Eds., ACM Press, pp. 413–422.
14. Gordon, S. D., and Katz, J. Partial fairness in secure two-party computation. In *Advances in Cryptology – EUROCRYPT 2010* (May 2010), H. Gilbert, Ed., vol. 6110 of *Lecture Notes in Computer Science*, Springer, pp. 157–176.
15. Halevi, S., Lindell, Y., and Pinkas, B. Secure computation on the web: Computing without simultaneous interaction. In *Advances in Cryptology CRYPTO 2011*, vol. 6841 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2011, pp. 132–150.
16. Han, S., and Ng, W. K. Preemptive measures against malicious party in privacy-preserving data mining. In *SIAM International Conference on Data Mining* (2008), pp. 375–386.
17. Harkavy, M., Tygar, J. D., and Kikuchi, H. Electronic auctions with private bids. In *Proceedings of the 3rd conference on USENIX Workshop on Electronic Commerce - Volume 3* (Berkeley, CA, USA, 1998), USENIX Association.
18. Harnik, D., Ishai, Y., Kushilevitz, E., and Nielsen, J. B. OT-combiners via secure computation. In *TCC 2008: 5th Theory of Cryptography Conference* (Mar. 2008), R. Canetti, Ed., vol. 4948 of *Lecture Notes in Computer Science*, Springer, pp. 393–411.
19. Huang, Y., Evans, D., Katz, J., and Malka, L. Faster secure two-party computation using garbled circuits. In *USENIX Security* (2011).
20. Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO 2003* (Aug. 2003), D. Boneh, Ed., vol. 2729 of *Lecture Notes in Computer Science*, Springer, pp. 145–161.
21. Jarecki, S., and Shmatikov, V. Efficient two-party secure computation on committed inputs. In *Advances in Cryptology – EUROCRYPT 2007* (May 2007), M. Naor, Ed., vol. 4515 of *Lecture Notes in Computer Science*, Springer, pp. 97–114.
22. Joan Feigenbaum, Benny Pinkas, R. R., and Saint-Jean, F. Secure computation of surveys. In *EU Workshop on Secure Multiparty Protocols* (2004).
23. Kamara, S., Mohassel, P., and Raykova, M. Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272, 2011.
24. Katz, J., and Malka, L. Constant-round private function evaluation with linear complexity. In *Advances in Cryptology – ASIACRYPT 2011* (Dec. 2011), Lecture Notes in Computer Science, Springer, pp. 556–571.

25. Kolesnikov, V., and Schneider, T. Improved garbled circuit: Free XOR gates and applications. In *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II* (July 2008), vol. 5126 of *Lecture Notes in Computer Science*, Springer, pp. 486–498.
26. Kolesnikov, V., and Schneider, T. A practical universal circuit construction and secure evaluation of private functions. In *FC 2008: 12th International Conference on Financial Cryptography and Data Security* (Jan. 2008), G. Tsudik, Ed., vol. 5143 of *Lecture Notes in Computer Science*, Springer, pp. 83–97.
27. Lindell, Y., and Pinkas, B. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology – EUROCRYPT 2007* (May 2007), M. Naor, Ed., vol. 4515 of *Lecture Notes in Computer Science*, Springer, pp. 52–78.
28. Lindell, Y., and Pinkas, B. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology 22*, 2 (Apr. 2009), 161–188.
29. Lindell, Y., and Pinkas, B. Secure two-party computation via cut-and-choose oblivious transfer. In *TCC 2011: 8th Theory of Cryptography Conference* (Mar. 2011), Y. Ishai, Ed., vol. 6597 of *Lecture Notes in Computer Science*, Springer, pp. 329–346.
30. Lopez-Alt, A., Tromer, E., and Vaikuntanathan, V. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *44th Annual ACM Symposium on Theory of Computing* (2012), ACM Press, pp. 1219–1234.
31. Mohassel, P., and Franklin, M. Efficiency tradeoffs for malicious two-party computation. In *Public Key Cryptography - PKC 2006*, vol. 3958 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 458–473.
32. Naor, M., and Pinkas, B. Efficient oblivious transfer protocols. In *12th Annual ACM-SIAM Symposium on Discrete Algorithms* (Jan. 2001), ACM-SIAM, pp. 448–457.
33. Naor, M., Pinkas, B., and Sumner, R. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce* (New York, NY, USA, 1999), ACM, pp. 129–139.
34. Nielsen, J. B., Nordholt, P. S., Orlandi, C., and Burra, S. S. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – Crypto 2012*, pp. 682–700.
35. Nielsen, J. B., and Orlandi, C. LEGO for two-party secure computation. In *TCC 2009: 6th Theory of Cryptography Conference* (Mar. 2009), O. Reingold, Ed., vol. 5444 of *Lecture Notes in Computer Science*, Springer, pp. 368–386.
36. Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. Secure two-party computation is practical. In *Advances in Cryptology – ASIACRYPT 2009* (Dec. 2009), M. Matsui, Ed., vol. 5912 of *Lecture Notes in Computer Science*, Springer, pp. 250–267.
37. Shelat, A., and Shen, C.-H. Two-output secure computation with malicious adversaries. In *Advances in Cryptology – EUROCRYPT 2011* (May 2011), K. G. Paterson, Ed., vol. 6632 of *Lecture Notes in Computer Science*, Springer, pp. 386–405.
38. Valiant, L. Universal circuits (preliminary report). In *STOC* (1976), ACM Press, pp. 196–203.
39. Yao, A. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science* (Oct. 1986), IEEE Computer Society Press, pp. 162–167.