Salus: A System for Server-Aided Secure Function Evaluation

Seny Kamara * Payman Mohassel † Ben Riva ‡

Abstract

Secure function evaluation (SFE) allows a set of mutually distrustful parties to evaluate a function of their *joint* inputs without revealing their inputs to each other. SFE has been the focus of active research and recent work suggests that it can be made practical. Unfortunately, current protocols and implementations have inherent limitations that are hard to overcome using standard and practical techniques. Among them are: (1) requiring participants to do work linear in the size of the circuit representation of the function; (2) requiring all parties to do the same amount of work; and (3) not being able to provide complete fairness.

A promising approach for overcoming these limitations is to augment the SFE setting with a small set of *untrusted* servers that have no input to the computation and that receive no output, but that make their computational resources available to the parties. In this model, referred to as *server-aided* SFE, the goal is to tradeoff the parties' work at the expense of the servers. Motivated by the emergence of public cloud services such as Amazon EC2 and Microsoft Azure, recent work has explored the extent to which server-aided SFE can be achieved with a *single* server.

In this work, we revisit the sever-aided setting from a practical perspective and design single-server-aided SFE protocols that are considerably more efficient than all previously-known protocols. We achieve this in part by introducing several new techniques for garbled-circuit-based protocols, including a new and efficient input-checking mechanism for cut-and-choose and a new pipelining technique that works in the presence of malicious adversaries. Furthermore, we extend the server-aided model to guarantee fairness which is an important property to achieve in practice.

Finally, we implement and evaluate our constructions experimentally and show that our protocols (regardless of the number of parties involved) yield implementations that are 4 and 6 times faster than the most optimized two-party SFE implementation when the server is assumed to be malicious and covert, respectively.

Keywords: secure computation, server-aided computation, multi-party computation, cloud computing

1 Introduction

Secure function evaluation (SFE) allows a set of n parties, each with a private input, to securely and jointly evaluate an n-ary function f over their inputs. Roughly speaking, a SFE protocol guarantees (at least) that: (1) the parties will not learn any information from the interaction other than their output and what is inherently leaked from it; and (2) that the functionality was computed correctly. SFE is useful in any setting where mutually distrustful parties need to cooperate but do not wish to reveal their inputs to each other. This often occurs in practice, for example, during negotiations, auctions, data mining, and voting.

*Microsoft Research. senyk@microsoft.com

†University of Calgary. pmohassel@cspc.ucalgary.ca.

‡Tel Aviv University. benriva@tau.ac.il. Work done at Microsoft Research.

Towards practical SFE. Early work on SFE—and the more general notion of multi-party computation (MPC)—focused on feasibility results; that is, demonstrating that *every* function can be computed securely [57, 58, 24, 10]. Motivated by these results, much of the work in SFE focused on improving the security definitions [44, 9], on strengthening the adversarial models, on decreasing the round and communication complexity and on improving efficiency [43, 45, 7, 38, 41, 4, 6, 52]. The special case of two-party SFE (2SFE) in particular has been improved extensively with more efficient semi-honest-to-malicious compilation techniques [43, 45, 56, 38, 41, 35, 40, 55], garbled circuit constructions and implementations [49, 35, 52, 29, 42], and oblivious transfer protocols [48, 30, 50]. In fact, recent work on *practical* SFE has even led to the design and implementation of several SFE/MPC frameworks such as Fairplay [43, 4], VIFF [15], Sharemind [5], Tasty [28], HEKM [29] and VMCrypt [42].

Limitations of standard SFE. While these advances have resulted in great improvements in efficiency and have effectively taken SFE from a purely theoretical idea to a potentially practical technology, there are inherent limitations to these optimizations. Indeed, in all current implementations—even with the state-of-the-art protocols and optimizations—the parties' running time is *linear in the size of the circuit* representation of the function they wish to evaluate.

Unfortunately, the circuit representation of most functions of practical interest can be quite large. For example, note that the circuit for evaluating the AES function requires roughly $30,000$ gates, while a circuit that evaluates the edit distance between two 50 character strings requires about $250,000$ gates (see Section 4.2). It is easy to think of interesting functionalities with circuits that go up to billions of gates.

An additional limitation of standard SFE is that the workload is *symmetric* in the sense that each party has to do the same amount of computation. In many real-world applications, however, some parties are weaker than others. Consider, e.g., a mobile device that retrieves location-based information from a server; or a PC that searches a distributed database hosted by a large-scale cluster of machines. In each case, SFE would allow the parties involved to perform the computation while keeping their inputs (i.e., the location and the search term) private. Unfortunately, standard protocols— all of which require a symmetric amount of work—are ill-suited for these settings since the size of both parties' inputs is bounded by the resources available to the weakest device. In other words, the mobile device would only be able to search through a small amount of location data, and the PC could only search through a small fraction of the database.

Finally, achieving *fairness* is not always possible with standard SFE. Roughly speaking, fairness ensures that either all the parties learn the output of the function being evaluated or none will. This is crucial in many real-world applications such as auctions, electronic voting, or collective financial analysis, where a dishonest participant should not be able to disrupt the protocol if it is not satisfied with the outcome of the computation. In 1986, Cleve showed that complete fairness is impossible in general, unless the majority of the players are honest [12]. A number of constructions try to achieve fairness for a specific class of functionalities [27], or consider limited (partial) notions of fairness instead [51, 19, 26].

Server-aided SFE. Several approaches to SFE have been proposed to address the inherent limitations outlined above and to achieve a *sub-linear* amount of work (in the size of the circuit representation of the function). Of course, one possibility is to design special-purpose protocols, i.e., protocols that work for a single functionality or a class of functionalities (see, e.g., [46, 25]). Another approach, referred to as *server-aided* or *cloud-assisted* SFE, augments the standard setting with a small set of servers that have no inputs to the computation and receive no output but that make their computational resources available to the parties. In this paradigm, the goal is to tradeoff the parties' work at

the expense of the servers. Server-aided SFE with two or more servers has been considered in the past [16, 17] and even deployed in practice [6].

Currently, there are two approaches to designing single-server-aided SFE protocols: (1) combining fully-homomorphic encryption (FHE) [21] with a proof system [3, 1]; or (2) using Yao's garbled circuit technique [57, 58]. The latter was proposed by Feige, Killian and Naor [18] in the semi-honest model and recently formalized and extended to stronger models by Kamara, Mohassel and Raykova [32]. The FHE-based approach is asymptotically optimal, but currently only of theoretical interest. The garbled circuit approach is slightly less efficient asymptotically but yields protocols that are efficient in practice. In addition, it can benefit from almost all the optimizations discussed above so it is complementary to previous work on practical SFE.

While the garbled-circuit approach yields practical protocols, we note that it is in a weaker adversarial model than the FHE-based solutions. In standard SFE it is assumed that if two or more parties are dishonest, they will collude. The protocols of [18] and [32], however, are only secure in settings where some of the dishonest parties *do not collude*. There are many settings in practice where collusion does not occur, e.g., due to physical restrictions, legal constraints and/or economic incentives. In a server-aided setting where the server is a large cloud provider (e.g., Amazon, Google or Microsoft), it is reasonable—given the consequences of legal action and bad publicity—to assume that the server will not collude with the parties.

Another limitation of the garbled circuit approach is that it yields protocols where at least one of the parties (other than the server) has to do linear work (throughout we will assume this party to be the first party P_1). It is shown in [32] that any non-interactive single-server-aided protocol without this limitation would imply a secure non-interactive delegated computation scheme which, currently, we only know how to construct based on FHE.

Deploying server-aided SFE. Even with the limitations outlined above, we believe that the garbled-circuit approach to single-server-aided SFE could be useful in many real-world scenarios. Consider, for example, the case of an enterprise (e.g., a small-to-medium-sized one) that may want to use SFE to provide better privacy for itself and its customers. The limitations of standard SFE outlined above would make this completely impractical—especially if the enterprise's clients are resource-constrained devices such as smartphones or tablets.

A better approach would be to use server-aided SFE where the server is instantiated by a public cloud (e.g., Amazon EC2, Windows Azure or Google Compute Engine), the enterprise plays the role of P_1 and its clients play the roles of the remaining parties. Note that such a setup is very natural in the setting of cloud computing since many business's already make use of public clouds to deliver services to their own clients (e.g., Netflix, Instagram) and since they have significantly more resources than their clients.

1.1 Background on Garbled Circuits

We make use of several cryptographic building blocks in our protocols, including garbled circuits, oblivious transfer (OT) [53] and secret sharing [54]. We omit formal descriptions of these building blocks due to lack of space and refer the reader to [22]. However, we briefly review garbled circuits before describing our contributions.

Yao's garbled circuit technique [57, 58] transforms circuits in such a way that they can be evaluated on encrypted inputs. We refer the reader to [39] for details on Yao's garbled circuit construction. At a very high level, a garbling scheme consists of: (1) a circuit garbling procedure that transforms a circuit into a garbled circuit consisting of garbled gates; (2) an input garbling procedure that generates a set of input labels; (3) an evaluation procedure that evaluates a garbled circuit over a set of input labels,

resulting in a set of output labels; and (4) a decoding procedure that recovers the output from the output labels.

The main security property provided by garbled circuits is *input privacy*, which guarantees that no information about the inputs is revealed by the evaluation and decoding procedures beyond what can be inferred from the output. Our protocols will rely on a slightly different property called *input/output* privacy which guarantees that no information about the inputs or outputs are revealed by the evaluation procedure (these properties are implied by the security proof of [39]). Another useful property of garbled circuits is *unforgeability* which, informally, guarantees that an incorrect evaluation can be detected with all but negligible probability. This property has also been noted and used in works as early as [49], but pointed out more explicitly in [20].

Cut-and-choose and input-consistency. A difficulty that often comes up when designing protocols based on garbled circuits is verifying whether a circuit was garbled correctly (this occurs when adversaries can be malicious). Several mechanisms exist to address this but the most efficient is *cut-and-choose* [47, 43, 45, 34, 56, 38]. With cut-and-choose, the garbler starts by constructing many garbled circuits. The evaluator chooses a random subset of these circuits and verifies their correctness by asking the garbler to reveal the secrets it used. If the verification goes through, the evaluator is left with several unopened garbled circuits and, with high probability, most of them are properly constructed (otherwise at least one malconstructed garbled circuit would have been detected during verification). The evaluator then evaluates the remaining garbled circuits and outputs the majority value.

This last step, however, introduces new problems and to avoid subtle attacks the evaluator has to check that the garbler used the same inputs for all the remaining circuits. This input checking step can be handled using several techniques. Mohassel and Franklin [45] and Lindell and Pinkas [38] extend the cut-and-choose technique to cover the input labels as well. Unfortunately, this requires a quadratic (in the security parameter) number of commitments. Another approach is to use specially-designed zero-knowledge proofs [40, 55] which, under specific number-theoretic assumptions, require only a linear number of exponentiations.

The techniques of [45] and [38] are extended to the server-aided setting in [32], where an input checking mechanism is described that requires a quadratic number of commitments.

Pipelined execution. Finally, since circuits can grow very large, garbling and evaluating them in memory can be expensive. Several implementations, therefore, *pipeline* the generation and evaluation of garbled circuits [31, 28, 29, 42] by having the garbler send (or stream) the garbled gates immediately after generating them and having the evaluator evaluate (or verify) them on the fly. Using this approach, the parties store in memory only the intermediate wires needed for the rest of the evaluation. This leads to very efficient implementations since the parties only need to store intermediate values and garbled gates on disk. Moreover, it improves the latency of the protocol since the garbler and the evaluator can operate simultaneously. Previous work, however, has only shown how to pipeline garbled circuits in the presence of semi-honest adversaries.

1.2 Our Contributions

Secure function evaluation is an important and powerful cryptographic primitive and many of its underlying techniques, such as garbled circuits, oblivious transfer and secret sharing, are important in their own right. As such, SFE and the underlying primitives that enable it have a wide array of applications and if made practical could have a large impact on the design of secure and privacy-preserving technologies.

Today, the server-aided approach seems to be one of the most promising directions for scaling SFE and overcoming the inherent limitations of the optimization techniques discovered in previous work. In addition, given the recent emergence of public clouds, such as Amazon EC2 and Microsoft Azure, server-aided SFE can be implemented broadly and at low-cost.

In this work, we revisit the server-aided setting from a more *practical* point of view and explore the extent to which we can take advantage of this model and design protocols that are more efficient and practical than what was previously known [18, 32]. Our findings are quite positive as we make the following contributions.

Fairness. We extend the server-aided model to provide *fairness*, which guarantees that either all parties will receive their outputs or none will. Fairness is an important property to achieve in the context of practical SFE as it provides an incentive for parties not to abort the protocol after receiving their outputs.

A generic transformation. We give a black-box transformation from any two-party SFE protocol secure against malicious adversaries to a multi-party server-aided SFE protocol. This yields a generic compiler for designing efficient server-aided SFE protocols and provides a point of comparison for custom-designed constructions.

Efficient server-aided protocols. We design two new single-server-aided multi-party SFE protocols. Both are based on garbled circuits, achieve fairness and are more efficient than any of the previously-known protocols [18, 32], or the generic construction mentioned above. In addition, the efficiency of the first protocol can be increased if it is used in the presence of a *covert* server (i.e., a server that wants to cheat but does not want to be caught) [2].

One of the reasons behind the improved efficiency is a new input checking mechanism for the cut-and-choose step that only requires a *linear* number of commitments (in the total length of the inputs), as opposed to previously-known alternatives that either require a quadratic number of commitments (in the security parameter) or a linear number of exponentiations.

In settings where the total input length is relatively small compared to the square of the security parameter (which is roughly $17,000$) our approach is significantly more efficient. For example, for the AES circuit our construction reduces the total size of the commitments from more than 100MB (in the quadratic case) to around 1MB.

Pipelining with malicious adversaries. We introduce a new pipelining technique that works in the presence of a *malicious* adversary. As discussed above, previous solutions were only applicable to the semi-honest setting and did not work when considering malicious adversaries. We note that in independent and concurrent work, Kreuter, shelat and Shen [36] describe an approach similar to ours.

Experimental evaluation. We implement our protocols and evaluate their efficiency empirically over two large circuits: (1) the AES circuit which has roughly $30,000$ gates; and (2) the edit distance circuit for 50 character strings of 8 bits each which has roughly $250,000$ gates. Naturally, all parties but P_1 have to do very little work. Interestingly, however, our experiments show a significant reduction in the total running times—even for the server and P_1. For AES, our first protocol (which is secure against a covert server) is roughly $6\times$ faster than the covert protocol of [52]. Our second protocol (which secure against a malicious server), is roughly $4\times$ faster than the most optimized 2SFE implementation [55]. Moreover, a nice property that our experiments illustrate is that the running time of our second protocol is almost independent of the number of the parties involved.

2 Model and Definitions

Practical server-aided SFE with a single server has only been achieved in certain specific adversarial models. In particular, as shown in [32], the garbled-circuit-based protocol of Feige, Killian and Naor from [18] is a secure sever-aided SFE protocol against a set of non-colluding semi-honest adversaries, that is, adversaries that follow the protocol and are independent in the sense that they do not share any information before or after the protocol execution. [32] also gives protocols that are secure in the presence of *non-cooperating* adversaries which, roughly speaking, are adversaries that deviate from the protocol but do not send information other than what prescribed by the protocol (note that a non-cooperating adversary is *stronger* than a semi-honest adversary).

A natural question, therefore, is whether these relaxations of the adversarial model are necessary in order to achieve practical server-aided SFE and all the advantages it provides, such as asymmetric efficiency (i.e., different parties needing different amounts of resources) and sub-linear work.

Asymmetric efficiency in the standard model? Consider a solution that does not make use of the relaxations described above. In particular, one might attempt to design an efficient server-aided protocol between parties $(P_1 \ldots, P_n)$ and a server S, such that: (1) a subset of the parties do sub-linear work; and (2) the server and the remaining parties do work that is polynomial in the size of the circuit. Such a protocol with security in the *standard* adversarial model, however, would yield a 2SFE protocol with low communication and computation for one party [1] which, currently, can only be constructed based on FHE [14].

Server-aided SFE from any two-party SFE. A second promising attempt (and a successful one) is to take advantage of the fact that the server and P_1 are never simultaneously malicious. With this assumption in place, one can indeed design practical protocols wherein all the parties but P_1 perform very little work (only proportional to their own input). The idea is as follows: the players $(P_2, \ldots, P_n)$ share their inputs between S and P_1, and let them run a general-purpose 2SFE protocol (with security against malicious adversaries) for computing the desired function on the players' inputs. This approach is promising but for it to work one needs to enhance the 2SFE protocol with mechanisms to convince the players that: (1) their real inputs were used (note that the security of 2SFE does not imply this); and (2) the output of the 2SFE is delivered back to them (2SFE guarantees output correctness but not honest delivery of the output to P_2 through P_n).

We now describe an efficient solution that addresses both issues, and works with *any* general-purpose 2SFE protocol with security against malicious adversaries. This, of course, is the most general case one can hope for in the context of server-aided SFE so we get a positive feasibility result that 2SFE implies server-aided SFE—though perhaps not with optimal efficiency.

Recall that we have parties $(P_1, \ldots, P_n)$, each with a secret input x_i, and a server S with no input or output. Let C be the circuit they wish to evaluate. The high-level idea of the reduction is as follows: the parties $(P_2, \ldots, P_n)$ share their inputs between S and P_1 who run the 2SFE protocol (with security against malicious adversaries) to evaluate the circuit C^{Sh} computes $\mathrm{C}(x_1^0 \oplus x_1^1, \ldots, x_n^0 \oplus x_n^1)$, where x_i^0 and x_i^1 are the shares of x_i. This solution is not sufficient, however, since the 2SFE protocol cannot prevent malicious S and P_1 from changing their inputs. Similarly, the party that learns the output of the 2SFE can simply lie about it to the other parties.

To solve these problems, we make use of a one-time message authentication code (MAC) in the 2SFE evaluation. To verify the outputs, each party P_j picks at random two l-bit strings v_j^0 and v_j^1.

[1] Given such a server-aided SFE protocol one can construct a standard two-party protocol by having the first party simulate the subset of the parties who perform sub-linear work and having the second party simulate the server S and the remaining parties.

Now, instead of evaluating the circuit C they evaluate the circuit C^{R} that receives $\mathbf{x}$ and the $n-1$ pairs $(v_j^0, v_j^1)_{j\in[n-1]}$, evaluates $\mathrm{C}(\mathbf{x})$ and outputs the values $v_2^{\mathrm{C}(\mathbf{x})}, \ldots, v_n^{\mathrm{C}(\mathbf{x})}$. The parties then share their inputs (including the pairs $(v_j^0, v_j^1)_{j\in[n-1}$) between P_1 and S who run the 2SFE protocol with the circuit $\mathrm{C}^{\mathsf{Sh}} \circ \mathrm{C}^{R}$. P_1 / S send to party P_j the value $v_j^{\mathrm{C}(\mathbf{x})}$. P_j verifies that the string he receives is one of v_j^0 and v_j^1 and determines the right output based on that. Note that this technique increases the input length by $2ln$ for each bit of the output. This overhead could be decreased by a factor of n by using only a single pair (v^0, v^1) that is generated obliviously by all parties. We omit the details of this optimization.

A similar but slightly more sophisticated technique can be used for the verification of the inputs. Let b be a bit of a share that party j sent to P_1 in the above protocol. The process we explain next needs to be repeated for every such b. Party P_j picks two l-bit strings k_0 and k_1 at uniformly at random, shares them between P_1 and S, and sends k_b to P_1. Now, P_1 and S execute a 2SFE protocol where, P_1's input is the shares of k_0, k_1 and k_b, and S's input is its shares of k_0, k_1. The circuit they evaluate is a circuit the reconstructs k_0, k_1 from the given shares, checks to see if k_b is equal to one of the keys (if not it aborts), and uses the value of b as the input for the circuit $\mathrm{C}^{\mathsf{Sh}} \circ \mathrm{C}^{\mathsf{R}}$. Note that in order to flip the bit b, P_1 has to guess S's share of k_{1-b}, and assuming we use the XOR of random l-bit values for secret sharing, this happens with only negligible probability in l. Of course, the same technique should be repeated for each bit of the shares held by P_1 and S, therefore increasing the total input length by a factor of $2l$.

We also point out that the same technique can be used to design a *multi*-server-aided SFE protocol from standard multi-party SFE in a black-box way. One can simply have the parties share their input between k servers, and run a k-party SFE protocol between them that evaluates a circuit similar to the one described above. The security of the multi-party SFE protocol will imply the security of the multi-server-aided SFE protocol as long as at least one of the servers is not malicious.

Better efficiency than 2SFE? An important question is whether the server-aided setting can lead to more efficient protocols than the standard setting or whether the best we can hope for is to achieve the complexity of existing 2SFE protocols. Unfortunately, the latter is indeed the case *in general*, i.e., if we insist on security with respect to all possible adversarial settings. Suppose, for example, that we require the server-aided protocol to be secure in the presence of either: (1) a malicious server and a single malicious party (say P_t); or (2) an honest server and all but one malicious parties. The overall complexity of such a protocol (even if carefully optimized) will always be greater than the complexity of the most efficient 2SFE protocol with security against malicious adversaries.

To see why, we sketch how one can use such a protocol to construct a 2SFE protocol secure against malicious adversaries. Let A and B be the two parties who want to engage in a 2SFE protocol. A runs the server-aided protocol simulating the server and P_t with A's input, and B runs the protocol simulating the rest of the parties (and sharing his input between them). If A is malicious, the security of the server-aided protocol against a malicious server and a malicious P_t guarantees security. If B is malicious, the security of the server-aided protocol against an honest server and a colluding set of all but one malicious parties (P_t in this case) implies security for the 2SFE protocol.

On a positive note, we show that if we restrict ourselves to certain adversarial settings then we can indeed do better than 2SFE. More precisely, we show that by only requiring security in the presence of a *non-cooperative* server, we can do much better.

2.1 Formal Model

We recall the ideal/real-model security definition for MPC in the presence of non-cooperative adversaries presented in [32]. At a high level, the definition compares the real-model execution of a protocol for computing an n-party function f to the ideal-model evaluation of f by a trusted party in the presence of m independent adversaries $(\mathcal{A}_1, \ldots, \mathcal{A}_m)$ that are assumed not to collude.

Non-collusion in MPC. The standard adversarial models for MPC include: (1) *semi-honest* adversaries which follow the protocol but attempt to learn extra information from their view of the execution; and (2) *malicious* adversaries which can deviate arbitrarily from the protocol. The recently proposed notion of *non-cooperative* adversaries [32] captures adversaries that may deviate from the protocol but that do not share any information that is not prescribed by the protocol.

Definition 2.1 (Non-cooperative adversary [32])**.** *An adversary $\mathcal{A}_i$ is non-cooperative with respect to adversary $\mathcal{A}_j$ if the messages $\mathcal{A}_i$ sends to $\mathcal{A}_j$ reveal no information about $\mathcal{A}_i$'s private values (i.e., its coins and input) to $\mathcal{A}_j$ beyond what can be inferred from $\mathcal{A}_j$'s output $f_j(\mathbf{x})$.*

Note that the notion of non-cooperation only restricts the information revealed by $\mathcal{A}_i$'s messages and does not imply that $\mathcal{A}_i$ is semi-honest. Indeed, $\mathcal{A}_i$ could deviate from the protocol without revealing any information to $\mathcal{A}_j$ about its private values, e.g., by garbling a function $f' \neq f$ when required to garble f.

2.2 Security Definition

Our security definition is similar to the one presented in [32] with the exception that it guarantees fairness and handles the case when the server is covert. (See [22] for more details about the idea-model/real-model security for MPC.) At a high level, fairness is guaranteed by modifying the behavior of the trusted party in the ideal-model execution so that it sends $\perp$ to all parties if any party chooses to abort (note that the fairness guarantee does not extend to the server). We capture covertness using the *explicit cheat* formulation of [2] which augments the ideal-model execution by allowing a covert adversary $\mathcal{A}$ to send a *cheat* instruction to the trusted party. Upon receiving this instruction, the trusted party sends $\mathcal{A}$ all the inputs and takes one of two possible actions: with probability ϵ it discloses to all parties that $\mathcal{A}$ cheated and with probability $1 - \epsilon$ it does not.

Real-model execution. The real-model execution of protocol Π takes place between parties $(P_1, \ldots, P_n)$, server P_{n+1} and adversaries $(\mathcal{A}_1, \ldots, \mathcal{A}_{m+1})$, where $m \leq n$.

At the beginning of the execution, each party $(P_1, \ldots, P_n)$ receives its input x_i, a set of random coins r_i, and an auxiliary input z_i while the server P_{n+1} receives only a set of random coins r_{n+1} and an auxiliary input z_{n+1}. Each adversary $(\mathcal{A}_1, \ldots, \mathcal{A}_m)$ receives an index $i \in \mathrm{I}$ that indicates the party it corrupts, while adversary $\mathcal{A}_{m+1}$ receives a set of indices that indicate the parties it will corrupt (this captures the fact that these parties collude).

For all honest parties P_i, let $\textsc{out}_i$ denote its output and for all corrupted parties P_i, let $\textsc{out}_i$ denote its view during the execution of Π. The ith partial output of a real-model execution of Π between parties $(P_1, \ldots, P_{n+1})$ in the presence of adversaries $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_{m+1})$ is defined as

$$\textsc{real}^{(i)}(k, \mathbf{x}; \mathbf{r}) \stackrel{def}{=} \left\{\textsc{out}_j : j \in \mathrm{H}\right\} \cup \textsc{out}_i.$$

where H denotes the set of honest parties and $\mathbf{r} = (r_1, \ldots, r_{n+1})$.

Ideal-model execution. In the ideal-model execution, all the parties interact with a trusted party that evaluates f. As in the real-model execution, the ideal execution begins with each party $(P_1, \ldots, P_n)$ receiving its input x_i, its coins r_i and an auxiliary input z_i, while the server P_{n+1} receives only its coins r_{n+1} and an auxiliary input z_{n+1}. Each party $(P_1, \ldots, P_n)$ sends x'_i to the trusted party, where $x'_i = x_i$ if P_i is semi-honest and x' is an arbitrary value if P_i is either malicious or non-cooperating. If any $x'_i = \perp$ or if the server sends an abort message, the trusted party returns $\perp$ to all parties. If the server P_{n+1} is covert, it can send a *cheat* instruction to the trusted party. Upon receiving this instruction, the trusted party tosses some coins and with probability $1 - \epsilon$ discloses the server's cheating (and the other parties output $\perp$) and with probability ϵ does not. In the latter case, the trusted party sends $(x'_1, \ldots, x'_n)$ to the server, receives $(y_1, \ldots, y_n)$ in response, and returns y_i to P_i. If the server did not send *cheat*, the trusted party returns $f_i(x'_1, \ldots, x'_n)$ to party P_i.

For all honest parties P_i, let OUT_i denote the output returned to P_i by the trusted party, and for all corrupted parties let OUT_i be some value output by P_i. The ith partial output of an ideal-model execution between parties $(P_1, \ldots, P_{n+1})$ in the presence of independent simulators $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_{m+1})$ is defined as

$$\text{IDEAL}^{(i)}(k, \mathbf{x}; \mathbf{r}) \stackrel{def}{=} \left\{\text{OUT}_j : j \in \text{H}\right\} \cup \text{OUT}_i.$$

where H denotes the set of honest parties and $\mathbf{r} = (r_1, \ldots, r_{n+1})$.

We now present our formal definition of security which, intuitively, guarantees that executing a protocol Π in the real model is equivalent to executing Π in an ideal model with a trusted party.

Definition 2.2 (Security)**.** *A n-party protocol Π securely computes f if there exists a set $\{\mathbf{Sim}_i\}_{i\in[m+1]}$ of polynomial-size transformations such that for all polynomial-size adversaries $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_{m+1})$, for all $\mathbf{x}$ and $\mathbf{z}$, and for all $i \in [m+1]$,*

$$\left\{\text{REAL}^{(i)}(k, \mathbf{x}; \mathbf{r})\right\}_{k\in\mathbb{N}} \stackrel{c}{\approx} \left\{\text{IDEAL}^{(i)}(k, \mathbf{x}; \mathbf{r})\right\}_{k\in\mathbb{N}}$$

where $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_{m+1})$ and $\mathcal{S}_i = \mathbf{Sim}_i(\mathcal{A}_i)$ and where $\mathbf{r}$ is chosen uniformly at random.

We now present a Lemma from [32] that we will use to prove security. Informally, it asserts that to prove a protocol Π secure in the presence of a non-cooperative adversary $\mathcal{A}_i$ it suffices to show that the protocol is secure in the semi-honest model (i.e., where all the adversaries follow the protocol) and that it is secure when $\mathcal{A}_i$ is malicious and all the other parties are honest.

Lemma 2.3 ([32])**.** *If a multi-party protocol Π between n parties $(P_1, \ldots, P_n)$ securely computes f in the presence of* (1) *independent and semi-honest adversaries and* (2) *a malicious $\mathcal{A}_i$ and honest $\{\mathcal{A}_j\}_{j\neq i}$; then Π is also secure in the presence of an adversary $\mathcal{A}_i$ that is non-cooperative with respect to all other semi-honest adversaries.*

3 Our Protocols

We describe our server-aided SFE protocols which are secure in the presence of a covert and malicious server, respectively. For simplicity of the description, we assume the output of the computation is a single bit. Extending it to more than one output bit is straightforward.

Notations and primitives. Denote by $\mathsf{inp}(P_i)$ the set of input wires for P_i's input bits and let m be the total number of input bits, i.e., $m = \sum |\mathsf{inp}(i)|$. Here we assume the output of the computation is a single bit, but both protocols can be easily adapted to handle multiple bit outputs.

Let C be a circuit that computes the function f. Let $\mathsf{GC}(\mathrm{C}; r)$ be an algorithm that, given a boolean circuit C and random coins r, outputs a garbled circuit $\widetilde{\mathrm{C}}$. Let $\mathsf{GI}(m; r)$ be an algorithm that, given an input length m and coins r, returns $2m$ input labels:

$$\mathbb{W} = \begin{pmatrix} w_1^0 & \dots & w_m^0 \\ w_1^1 & \dots & w_m^1 \end{pmatrix},$$

such that w_i^0 and w_i^1 are the labels of 0 and 1, respectively, for the ith input wire. If $\mathbf{x}$ is an m-bit string, we denote by $\mathbb{W}_{|\mathbf{x}}$ the label vector $(w_1^{x_1}, \dots, w_m^{x_m})$. Let $\mathsf{GO}(r)$ be an algorithm that, given coins r, returns two output labels (ω^0, ω^1) and let $\mathsf{Dec}(\omega; r)$ be a decoding algorithm that, given an output label ω and coins r, returns a bit b. Finally, let $\mathsf{Eval}(\widetilde{\mathrm{C}}, \mathbb{W}_{|\mathbf{x}})$ be an evaluation algorithm that, given a garbled circuit $\widetilde{\mathrm{C}}$ and a set of input labels $\mathbb{W}_{|\mathbf{x}}$, returns an output label ω. We require that for all circuits C and all coins r,

$$\mathsf{Eval}\Big(\mathsf{GC}\big(\mathrm{C}; r\big), \mathbb{W}_{|\mathbf{x}}\Big) = \omega^{f(\mathbf{x})},$$

and that for $b \in \{0, 1\}$, $\mathsf{Dec}\big(\omega^b; r\big) = b$, where $\mathbb{W} := \mathsf{GI}(m; r)$ and $(\omega^0, \omega^1) := \mathsf{GO}(r)$. For security, we also require input/output privacy which guarantees that a pair $(\widetilde{\mathrm{C}}, \mathbb{W}_{|\mathbf{x}})$ reveal no partial information about $\mathbf{x}$ and $f(\mathbf{x})$; and unforgeability which guarantees that an incorrect evaluation can be detected.

Our protocols make use of several standard cryptographic primitives, including pseudo-random functions, commitments, secret sharing and symmetric key encryption (see [23] and references therein for thorough discussion of their security property and pointers to instantiations). We will denote by $F_K(\cdot)$ a pseudo-random function with key K. Let $H(\cdot)$ be a one-way function (we use SHA, but we only need its one-wayness property); $\mathsf{Com}(m)$ be a commitment to message m; and $\mathsf{Enc}(k, m)$ be a (deterministic) symmetric encryption of a message m under a key K. Any standard instantiation of the above primitives works for us but following previous implementations [52], we use SHA-1 and SHA-256 as pseudo-random functions and use them to implement all other primitives (see Section 4.1). Share will denote the sharing algorithm of a n-out-of-n secret sharing scheme, i.e., $\mathsf{Share}(n, x)$ outputs n shares $(\sigma_1, \dots, \sigma_n)$ of x such that no partial information about x can be recovered unless *all* shares are held. Throughout, we will assume that sharing is instantiated with the simple XOR secret sharing scheme which, given an input x, returns n shares $(r_1, \dots, r_{n-1}, \bigoplus_{i=1}^{n-1} r_i \oplus x)$, where each r_i is a $|x|$-bit string chosen uniformly at random. Last, we denote by $[n]$ the set $\{1, \dots, n\}$.

3.1 Security Against a Covert Server

Our first protocol is fair and secure in the presence of a *covert* server which, roughly speaking, means that the server is dishonest but does not want to get caught. The covert adversarial model was introduced by Aumann and Lindell [2] and allows for more efficient protocols than the standard malicious model. Assuming that the server is covert (as opposed to fully malicious) seems natural in settings where there are strong incentives not to cheat. If the server is a large cloud provider (e.g., Amazon or Microsoft), this assumption is quite reasonable since the provider's reputation is at stake. We note that for our protocol, if the communication between all parties and the server is digitally signed, the parties can use the transcript as a proof that the server cheated.

As for fairness, we observe that although it is unachievable in the standard SFE setting (with a dishonest majority), it is achievable in the server-aided setting, hence providing a stronger security guarantee than standard SFE in this respect.

Recall the server-aided setting where a set of parties $(P_1, \dots, P_n)$ each with a private input and a server S with no input or output, want to collectively compute a function f over their private inputs. Let C be a Boolean circuit of size $|\mathrm{C}|$ that evaluates f and let $\mathbf{x} = (x_1, \dots, x_m)$ be a binary string

that represents the joint input of all the parties. Our construction makes use of garbled circuits and a distributed OT protocol. Next, we provide a high level overview of the protocol. A detailed description can be found in Figures 1 and 2.

Overview. S garbles a small number s_1 (e.g., 16) of circuits and sends them to P_1. It then shares the input labels for all the wires between all the parties (i.e., each party receives a share of each label of each wire) and sends them some secret information. Each party P_i communicates with all the other parties once and uses the secret information it received from S and the messages exchanged to generate the label corresponding to his input. This mechanism can be viewed as a special *distributed* OT protocol (where the server plays the role of the sender) that allows each party P_i to learn the label for its input x_i, without the server or the other parties $\{P_j\}_{j\neq i}$ learning anything about it.

P_1 then picks a circuit at random and asks the server to reveal the randomness it used to construct all the other circuits. This randomness also includes the coins used for the secret sharing of the labels. P_1 checks that this randomness is indeed correct and all the parties verify that their shares are derived from it. Finally, all the parties send their labels to P_1 who evaluates the remaining circuit and returns the result to all the parties.

Distributed OT. As discussed above, the distributed OT protocol allows the parties to privately retrieve their labels from the server, i.e., without the server or the other parties learning anything about them. We construct such a protocol based on black-box access to a PRF and *without invoking a standard OT protocol*.

The server holds two inputs a_0 and a_1, and one of the parties P_j holds a bit b. The distributed OT should let P_j learn a_b without revealing any additional information. The intuition behind the construction is as follows: the server secret shares a_0 and a_1 between all the parties, permuting each pair of shares randomly. The server sends to P_j the permutations that he used for all the parties. P_j asks for one share from each party according to the bit b and the random permutations he receives from the server. Note that the parties do not learn any information about b since they do not know the random permutations and P_j only learns one of a_0 and a_1 (assuming that at least one party does not cooperate with P_j). This solution, however, is not enough since if one of the shares is wrong, there is no way to determine if it is the server's fault or one of the parties. Thus, the server commits to all the shares and sends those commitments to the parties. Before requesting a share, P_j checks with the other parties that they received the same commitments, and only then proceeds with choosing the share.

Achieving fairness. Unfortunately, the protocol as described above does not provide fairness since P_1 can abort after seeing the output. We fix this by hiding the output from P_1 until all the parties receive the same output label. Consider a single output wire w_i. The output of the garbled circuit on this wire is a label and its translation to the real output bit is only known to S. After P_1 sends this label to all the parties, the server sends hashes of the two possible labels (one for the wire bit 0 and one for wire bit 1) in random order. The parties check that the hash of the label they have is equal to one of those hashes, and if so, they return an ACK message to S. After all the parties return an ACK, S sends the two labels to all the parties along with their mappings to the output value. Now each party can map its label to a corresponding output bit. One problem that arises in this protocol is that the server can change the output to whatever he wants by sending an incorrect mapping. We solve this by simply asking S to commit to the mappings in advance and having the parties check those commitments as part of the randomness revealing stage.

Setup and inputs: Each party P_i has an m_i-bit input while the server has no input. Let $m = \sum_{i\in[n]} m_i$. The server S holds a secret key K for a pseudo-random function F. s is a statistical security parameter. C is the circuit that computes f.

Distributed OT :

For all $\ell \in [s_1]$:

1. S computes $r_\ell := F_K(\ell)$. All the coins used by S for the ℓth circuit will be derived from r_ℓ,
2. S computes $\mathbb{W}^\ell := \mathsf{GI}(m; r_\ell)$ and for all $i \in [m]$, $(\sigma^0_{1,i}, \ldots, \sigma^0_{n,i}) \leftarrow \mathsf{Share}(n, w^0_i)$ and $(\sigma^1_{1,i}, \ldots, \sigma^1_{n,i}) \leftarrow \mathsf{Share}(n, w^1_i)$,
3. S then samples a $n \times m$ binary matrix $\mathbb{P}^\ell$ uniformly at random and generates the $n \times m$ matrix $\mathbb{S}^\ell$ defined as:
$$\mathbb{S}^\ell = \begin{pmatrix} \mathbb{P}^\ell_{11}\left[\sigma^0_{1,1}, \sigma^1_{1,1}\right] & \cdots & \mathbb{P}^\ell_{1m}\left[\sigma^0_{1,m}, \sigma^1_{1,m}\right] \\ \vdots & & \vdots \\ \mathbb{P}^\ell_{n1}\left[\sigma^0_{n,1}, \sigma^1_{n,1}\right] & \cdots & \mathbb{P}^\ell_{nm}\left[\sigma^0_{n,m}, \sigma^1_{n,m}\right] \end{pmatrix},$$
where $\mathbb{P}^\ell_{ij}\left[v^0, v^1\right] \stackrel{def}{=} \left(v^{\mathbb{P}^\ell_{ij}}, v^{1-\mathbb{P}^\ell_{ij}}\right)$,
4. S then constructs the $n \times m$ matrix $\mathbb{C}^\ell$ such that $\mathbb{C}^\ell_{ij} = \left(\mathsf{Com}\left(\mathbb{S}^\ell_{ij}[1]\right), \mathsf{Com}\left(\mathbb{S}^\ell_{ij}[2]\right)\right)$, where $\mathbb{S}^\ell_{ij}[a]$ for $a \in \{1, 2\}$ denotes the ath element of the pair stored at location ij of $\mathbb{S}^\ell$,
5. for all $i \in [n]$,
   (a) S sends the ith rows of $\mathbb{S}^\ell$ and $\mathbb{C}^\ell$ and the associated decommitments to P_i,
   (b) if the decommitments are invalid P_i accuses S and aborts,
   (c) for all $j \in \mathsf{inp}(P_i)$, S sends the jth column of $\mathbb{P}^\ell$ to P_i,
6. S sends to all parties $\mathrm{Q}^\ell_0 := \mathsf{Com}(\omega^0_\ell)$ and $\mathrm{Q}^\ell_1 := \mathsf{Com}(\omega^1_\ell)$, and, $H(\omega^0_\ell)$ and $H(\omega^1_\ell)$ permuted in a random order, where $(\omega^0_\ell, \omega^1_\ell) := \mathsf{GO}(r_\ell)$.

Cut-and-choose :

1. For all $\ell \in [s_1]$, S sends $\widetilde{\mathrm{C}}_\ell := \mathsf{GC}(\mathrm{C}; r_\ell)$ to P_1.
2. P_1 sends $e \stackrel{\$}{\leftarrow} [s_1]$ to S.
3. S sends $\{r_i\}_{i\in[s_1]-e}$ to P_1 who in turn sends it to all the parties.
4. All parties verify that all the values received from S in the previous steps were constructed properly from the appropriate randomness. If not, they accuse S and abort.

Input label reconstruction for P_i :

For all $j \in \mathsf{inp}(P_i)$

1. for all $i' \neq i$:
   (a) P_i sends $b_{i'j} := x_j \oplus \mathbb{P}^e_{i'j}$ to P'_i
   (b) $P_{i'}$ returns $\mathbb{S}^e_{i'j}[b_{i'j}]$ (recall that $P_{i'}$ received the i'th row from S in step 5(a) of th distributed OT phase).
2. P_i reconstructs $w^{x_j}_j$ using the n shares obtained in the previous steps.

Figure 1: Protocol 1 - Covert Server (Part 1)

Commitment consistency check :

For all $i \in [n]$, $j \in \mathsf{inp}(P_i)$ and $i' \neq i$:

1. S sends $\mathbb{C}^e_{i'j}$ to P_i,
2. P_i and $P_{i'}$ check that they have the same commitments (simply by sending them to each other). If not, they accuse S and abort. More precisely:
   (a) for all $j \in \mathsf{inp}(P_i) \cup \mathsf{inp}(P_{i'})$ they check that they both received the same commitments $\mathbb{C}^e_{ij}$ or $\mathbb{C}^e_{i'j}$ (depending on who owns wire j),
   (b) they check that they received the same Q^e_0 and Q^e_1,
3. $P_{i'}$ sends to P_i decommitments to $\mathbb{C}^e_{i'j}[b_{i'j}]$. If any decommitment is invalid, P_i accuses $P_{i'}$ and aborts.

Garbled circuit evaluation :

1. All the parties send their input labels for $\widetilde{\mathrm{C}}_e$ to P_1.
2. P_1 evalautes $\widetilde{\mathrm{C}}_e$ and returns the garbled output z to all the parties.

Revealing the output :

1. Each party P_i computes a hash of z and verifies that it matches one of the two hashes $H(\omega^0_e)$ and $H(\omega^1_e)$ the server sent earlier. If so, it sends an ACK message to the server.
2. After receiving an ACK messages from all players, S sends the decommitments to Q^e_0 and Q^e_1 to all parties.
3. Using the decommitments and z, each party can determine the output bit.

Figure 2: Protocol 1 - Covert Server (Part 2)

Asymptotic efficiency. Let s be a statistical security parameters, n the number of parties, and m be the combined length of all parties' inputs. With our protocol, S and P_1 work in time $O(s \cdot |\mathrm{C}| + smn)$ while the other parties work in time $O(sm)$, where for the specific values $s = 16$ (as suggested in [52]) we obtain a complexity of $O(16 \cdot |\mathrm{C}| + 16mn)$ and $O(16m)$, respectively. We emphasize that we only use *inexpensive* cryptographic primitives such as hash functions and commitments.

Security. We now turn to security and show in the following Theorem that our protocol is secure according to Definition 2.2.

Theorem 3.1. *The protocol fairly and securely computes the circuit* C *in the following two corruption scenarios:* (1) *The server is covert (but non-cooperative with respect to the parties), while all other parties are semi-honest,* (2) *the server is semi-honest, while all but one of the parties is malicious (but non-cooperative with respect to the server).*

Proof. Fairness is achieved because the server reveals the translation of the outputs only after all the parties confirm that they have received the same answer. Next we focus on a simulation-based proof of privacy and correctness.

For the above protocol, our server-aided security definition requires a simulation-based privacy and correctness guarantee in the following two scenarios: (1) the server S is *covert* and *non-cooperative*, while the parties are *semi-honest*; and (2) the server is *semi-honest*, and all but one of the parties are malicious. The malicious parties can collude between themselves.

Note that due to Lemma 2.3, we can divide the proof into three different claims: first we prove security when the parties and server are independent and semi-honest. Then, we prove security for

the case when the server is covert and the parties are honest, and finally we prove security in the case where the server is honest and all but one of the parties are malicious. The Lemma then extends these security guarantees to the case where the honest parties/server are replaced by semi-honest ones as mentioned above. We consider each claim separately.

Claim. The protocol securely computes the circuit C in presence of a semi-honest server $\mathcal{A}_S$, and semi-honest parties (all parties are independent).

We describe three independent simulators $\mathbf{Sim}_S$, $\mathbf{Sim}_1$ and $\mathbf{Sim}_i$, simulating the independent adversaries $\mathcal{A}_S$, $\mathcal{A}_1$ and $\mathcal{A}_i$ who corrupt the server, party P_1 and parties P_i (for $i > 1$) respectively.

- $\mathbf{Sim}_S$ simulates $\mathcal{A}_S$ as follows. The simulation in this case is quite straightforward. $\mathcal{A}_S$'s view in the real execution only consists of the random index e sent to it by P_1 and the final ACK messages from all parties at the end. $\mathbf{Sim}_S$ generates an identical view for $\mathcal{A}_S$ in the ideal execution. Note that since the parties are semi-honest there will be no abort.

- $\mathbf{Sim}_1$ receives P_1's input x_1 and sends it to the trusted party in order to receive $\mathrm{C}(\mathbf{x})$. It then randomly guesses the index e' and creates the correct garbled circuits $\widetilde{\mathrm{C}}_j$ for $j \in [s_1] - e'$, but a fake garbled circuit $\widetilde{\mathrm{C}}_{e'}$ that evaluates to $\mathrm{C}(\mathbf{x})$ on all inputs. When they reach the cut-and-choose step, $\mathcal{A}_1$ sends an index e. If $e \neq e'$, $\mathbf{Sim}_1$ rewinds the protocol and starts the process again. After an expected $O(1/s_1)$ times, its guess e' will be right one. At this point, $\mathbf{Sim}_1$ creates $\mathcal{A}_1$'s view in the ideal execution by sending it all the messages an honest server would (he can do so since the server has no inputs), and does the same on behalf of all the other parties, but using fake inputs for them (since he doesn't know their real inputs). The claim is that $\mathcal{A}_1$'s view in the real and ideal executions are indistinguishable. The two main differences in the views are that a fake garbled circuit $\widetilde{\mathrm{C}}_e$ is sent to $\mathcal{A}_1$ in the ideal execution. But as shown in previous work on Yao's garbled circuit construction [38], it is possible to efficiently generate a fake garbled circuit with the same output, such that the real and the fake circuits are computationally indistinguishable to $\mathcal{A}_1$. Another component of $\mathcal{A}_1$'s view that is different in the ideal execution is the fact that $\mathbf{Sim}_1$ uses fake inputs for other parties. Since $\mathcal{A}_1$ only sees random XOR shares of those inputs, however, this component of the views will be uniformly random and identical in both executions.

- $\mathbf{Sim}_i$'s description is very similar to $\mathbf{Sim}_1$ and is even simpler since $\mathcal{A}_i$ for $i > 1$ does not need to receive the complete garbled circuits but only commitments to outputs and shares of the garbled inputs.

This ends the proof of our first claim.

□

Claim. The protocol securely computes the circuit C in the presence of a covert server and honest parties.

For this claim, we only need to provide a simulator $\mathbf{Sim}_S$ for the adversary $\mathcal{A}_S$ but unlike the previous claim, $\mathcal{A}_S$ is not semi-honest and can actively cheat.

$\mathbf{Sim}_S$ plays the role of the honest parties using fake inputs for each. It receives garbled circuits, commitments and hash values from $\mathcal{A}_S$. It then plays the role of the honest P_1 by sending a random $e \in [s_1]$, checks to see if $\widetilde{\mathrm{C}}_j$ or any of the related commitments are not computed correctly for $j = e$ (or if $\mathcal{A}_S$ aborted). It then rewinds and sends a different random e. Note that in this stage the simulator

knows, for all circuits and commitments, whether they are correct or not. There are three cases now. First, if at least two circuits (or their commitments or hashes) are incorrect, it sends an abort message to the trusted party, simulates P_1 aborting and outputs whatever $\mathcal{A}_S$ does. Second, if exactly one of the circuits/commitments/hashes is incorrect, it sends the "cheat" instruction to the trusted party to notify it of being corrupted. If the trusted party discloses the cheating, $\mathbf{Sim}_S$ rewinds to the cut-and-choose step, chooses a value e where the server gets caught, simulates P_1 aborting, and outputs whatever $\mathcal{A}_S$ does. If the trusted party does not disclose the cheating, it rewinds to the cut-and-choose step, chooses e such that the server does not get caught, and outputs whatever $\mathcal{A}_S$ does. Third, if all circuits/commitments/hashes are correct, it sends an ACK message on behalf of each party to $\mathcal{A}_S$ and outputs what he does.

This ends the simulation. Note that if more than two bad circuits/commitments/hashes exists, the honest parties in the ideal execution and the real execution both abort. If there is exactly one bad circuit, this will happen in both models with probability $1 - 1/s_1$, and if everything is done correctly, both the real and the ideal executions finish successfully and with correct outputs for the honest parties.

□

Claim. The protocol securely computes the circuit C in presence of an honest server, and an all-but-one set of malicious parties.

Our final claim is for the case where all-but-one of the parties may be malicious and even collude. Consider the adversary $\mathcal{A}$ corrupting a subset of the parties. Without loss of generality, we assume that P_1 is among the corrupted parties. The reason is that the case where P_1 is not among the corrupted parties can easily be proved as a special case of the former.

One may wonder if the proof for this case is identical to the proof of the first claim in the case where P_1 is semi-honest. Unfortunately, however, a complication arises here that is not present in that case. Since P_1 is malicious, we need to extract his input during the simulation in order to obtain the output from the trusted party, and then use the output to create a consistent fake garbled circuit. But the input distribution stage does not take place until after the garbled circuits are sent to P_1. Hence, we need a slightly different simulation strategy.

Simulator **Sim** plays the role of the honest server S and at least one honest party P_i during the interaction with the rest of the parties. It starts by guessing e (the index of the evaluated circuit) and by preparing correct garbled circuits for all $\widetilde{\mathrm{C}}_i$ where $i \in [s_1] - \{e\}$. For $\widetilde{\mathrm{C}}_e$, it garbles a circuit that outputs P_1's first input bit (of course, with many dummy gates to make the garbled circuit indistinguishable from a valid circuit). The simulator runs the protocol until the cut-and-choose stage. If e was not selected by P_1, it rewinds the protocol and starts again. After an expected $O(1/s_1)$ times, its guess will be right. In that case, the simulation continues, and in the step in which P_1 (and the rest of the parties) ask P_i for their shares, the simulator learns the inputs of those parties (since it knows the permutations). The simulator then sends to the trusted party those inputs and receives the output of the computation.

Now, it changes the share that P_i has for P_1' s first input bit in such a way that P_1's garbled circuit evaluation will yield an output token that decodes to the correct output. Note that this can be done because we use XOR to share the input tokens. Indeed, the simulated P_i can flip P_1's input bit by simply XORing the shares with the input tokens it wants P_1 to recover. The commitments for those shares are recomputed and the protocol continues until the end, when the server and P_i send the same commitments to the other parties. Note that now $\mathcal{A}$ evaluates the circuit that returns the first bit of his input, and his first bit is the actual output bit. If P_1 returns an invalid output, he sends an abort to the trusted party, simulates honest P_i aborting and outputs what $\mathcal{A}$ does.

It is easy to show that the corrupted parties' view when interacting with the simulator is indistinguishable from their view in the real protocol: first, due to the security properties of Yao's garbled circuits, the fake circuit generated by the simulator is indistinguishable from a real one, and second, the corrupted parties' view of the honest P_i's input only consist of uniformly random shares and hence is identical in both executions.

□

The above three claims combined, complete the proof of security for our first server-aided SFE protocol with a covert server.

■

3.2 Security Against a Malicious Server

Our second protocol is secure against a malicious server and is described in detail in Figure 3. We now provide an overview.

Overview. In this protocol we assume the parties share randomness in the beginning. This can be achieved by simply running a simulatable coin tossing protocol [37, 33]. Such a protocol emulates the usual coin-flipping functionality in the presence of arbitrary malicious adversaries and allows a simulator who controls a single player to control the outcome of the coin flip.

Here, P_1 is the garbler and S is the evaluator. P_1 uses the shared randomness to generate s garbled circuits which it sends to the server S. To verify the correctness of the garbling step, S and P_1 execute a cut-and-choose protocol. At the end of the cut-and-choose, S is left with λ circuits, the majority of which are properly garbled (with high probability). Then, all the parties send the labels for their input wires to S (they can compute these labels using the shared randomness). Since this is done for multiple circuits, we have to ensure that each party uses the same input in all the circuits.

Input checking. There are several mechanisms to check and enforce input consistency [45, 38, 40, 55, 32] but we deviate from previous approaches and introduce a new mechanism that is more efficient. In particular, we require that for each wire $i \in [m]$, each party send to S the following two hashes permuted at random: $H\left(w^0_{1,i}|\cdots|w^0_{\lambda,i}\right)$ and $H\left(w^1_{1,i}|\cdots|w^1_{\lambda,i}\right)$, where $w^b_{j,i}$ is the input label for bit b of the ith wire of the jth circuit (for $j \in [\lambda]$). The server verifies that the hashes it received from different parties are the same. Assuming that at least one party is honest, this implies the hash was honestly computed. Then, given the labels for the ith input wire, S can compute their hash and verify that the result indeed matches one of the two previously accepted hashes for the same wire. If the check passes for all input wires, the server proceeds to the evaluation of the remaining λ circuits. At the end of the evaluation, S is left with λ output labels (the results of the λ evaluations). If the server directly sends these labels to the parties, however, it will leak additional information to them (as already pointed out in [32]).

We use a new technique for resolving this issue, that allows the server to output a single value that represents the majority output *without revealing any additional information.* This new technique is more efficient than the oblivious cut-and-choose technique of [32].

Asymptotic efficiency. The complexity of the protocol is as follows: let s be the security parameter (the number of garbled circuits), $\lambda < s$ be the number of circuits used for evaluation, n be the number of parties, and m be the total length of all the parties' inputs combined. S and P_1 work in time $O(s \cdot |\mathrm{C}| + sm)$ and the other parties work in time $O(\lambda m)$, where for the specific values $s = 132$ and

Setup and inputs: Each party P_i has an m_i-bit input while the server has no input. Let $m = \sum_i |m_i|$. All the parties share a secret key K. s and λ are statistical security parameters.

Setup shared randomness :

1. For all $\ell \in [s]$, the players compute $r_\ell = F_K(\ell)$ (these will be used to garble s circuits).
2. The players compute $\gamma_0 := F_K(s+1)$ and $\gamma_1 := F_K(s+2)$ (these will be used to decode the final output).

Circuit Garbling :

For all $\ell \in [s]$, P_1 sends $\widetilde{\mathrm{C}}_\ell := \mathsf{GC}(\mathrm{C}; r_\ell)$ to S.

Garbled Circuit Verification :

1. S picks a set $T \subset [s]$ of size $s - \lambda$ at random, and sends T to P_1.
2. P_1 sends to r_ℓ for all $\ell \in T$, to S.
3. For all $\ell \in T$, S checks that $\widetilde{\mathrm{C}}_\ell$ was created using r_ℓ. If so, it sends r_ℓ to all the parties who verify that it is equal to the randomness they computed earlier.

Input label transfer :

Let $E = [n] - T$ be the set of indices of non-verified circuits. Each party P_i computes $\mathbb{W}^\ell := \mathsf{GI}(m; r_\ell)$ for all $\ell \in E$. It then sends all its input labels to S. Denote by $w_{\ell,j}$ the label S receives for the jth input wire of the ℓth circuit.

Input label consistency check :

For $j \in [m]$:

1. All the players send the hash values $h_{w,b} = H(w^b_{\ell_1,j} | \cdots | w^b_{\ell_\lambda,j})$ for $b \in \{0,1\}$ in a random order, where $\ell_1, \ldots, \ell_\lambda \in E$.
2. S checks that it receives the same hash values from all the players.
3. S checks that one of the hashes equals $H(w_{\ell_1,j} | \cdots | w_{\ell_\lambda,j})$.

Garbled circuit evaluation :

S evaluates the circuits $\widetilde{\mathrm{C}}_\ell$ for all $\ell \in E$. Denote the output of $\widetilde{\mathrm{C}}_\ell$ by z_ℓ.

Majority output :

For all $\ell \in E$:

1. each party sends the two ciphertexts $\mathsf{Enc}(\omega^0_\ell, \gamma_0)$ and $\mathsf{Enc}(\omega^1_\ell, \gamma_1)$ to S permuted in a random order, where $(\omega^0_\ell, \omega^1_\ell) := \mathsf{GO}(r_\ell)$,
2. S checks that the pairs of encryptions it receives from all the players are identical and aborts otherwise,
3. S decrypts the two ciphertexts using z_ℓ and recovers γ_ℓ and γ'_ℓ.

S sends to all parties the value W that appears the most among the set $\{\gamma_\ell, \gamma'_\ell\}_{\ell \in E}$.

Output recovery :

Players output the bit b such that $\gamma_b = W$.

Figure 3: Protocol 2 - Malicious Server

$\lambda = 2s/5$ (as suggested in [55]) we get a complexity of $O(132 \cdot |\mathrm{C}| + 132m)$ and $O(52m)$, respectively. Again, we stress that we only use *inexpensive* cryptographic primitives.

Comparison to oblivious cut-and-choose. The oblivious cut-and-choose technique of [32] (executed between S and P_1) works in time $O(s \cdot |\mathrm{C}| + s^2 m)$ and each party P_i (for $i > 1$) works in time $O(\lambda^2 m_i)$ where m_i is the length of his own input. Hence, the price of checking input consistency is quadratic in the security parameter (alternatively, using the techniques of [40, 55] this can be done at the cost of a linear number of exponentiations). In many settings, the size of the input is not too large, and as such, it is more efficient to have the parties work in time that is proportional to the *total* input size than in time that is quadratic in the security parameter. Furthermore, our approach can be composed with the oblivious cut-and-choose technique so that a party P_i can either work in time that is quadratic in the security parameter (but only linear in his own input length), or add to the work of all other parties, by having them do additional work linear to P_i's input length.

Security. In the following Theorem, we show that our protocol is secure according to Definition 2.2.

Theorem 3.2. *The protocol fairly and securely computes the circuit* C *in the following two corruption scenarios: (1) The server is malicious (but non-cooperative with respect to the parties), while all other parties are semi-honest, (2) the server is semi-honest, while all but one of the parties is malicious (but non-cooperative with respect to the server).*

Proof. Fairness is achieved naturally here since the server reveals the same output to all the parties. Next we focus on a simulation-based proof of privacy and correctness.

Our server-aided security definition requires a simulation-based privacy/correctness guarantee in the following two scenarios: (1) the server S is *non-cooperative*, while the parties are *semi-honest*; and (2) the server is *semi-honest*, and all but one of the parties are malicious. The malicious parties can collude between themselves.

The proof has a very similar structure to that of the first protocol and contains three different claims. Lemma 2.3 ensures that proving the following three claims is sufficient to prove security in the above two scenarios.

Claim. The protocol securely computes the circuit C, in the presence of a semi-honest server $\mathcal{A}_S$, and semi-honest parties (all parties are independent).

As before, we describe three independent simulators $\mathbf{Sim}_S$, $\mathbf{Sim}_1$ and $\mathbf{Sim}_i$, simulating the independent adversaries $\mathcal{A}_S$, $\mathcal{A}_1$ and $\mathcal{A}_i$ corrupting the server, party P_1 and parties P_i (for $i > 1$) respectively.

- $\mathbf{Sim}_S$ simulates $\mathcal{A}_S$ in the following simple fashion: $\mathbf{Sim}_S$ plays the role of the semi-honest parties using fake inputs for each, and following the steps of the protocol. We need to show that this strategy produces a view that is indistinguishable from the real one. $\mathcal{A}_S$'s view in the real execution consists of the garbled circuits, the party's input labels for the unopened circuits, and randomly permuted hashes of the output labels. His view, however, does not include the output of the computation. By the security properties of Yao's garbled circuits, we have that the combination of the input labels and the garbled circuit are indistinguishable for any two inputs. Combined with the hiding property of the commitment scheme, and the fact that the hashes are randomly permuted, this implies that $\mathcal{A}_S$'s view is indistinguishable in the real and the ideal executions.

- $\mathbf{Sim}_1$ receives P_1's input x_1 and sends it to the trusted party in order to receive $\mathrm{C}(\mathbf{x})$. He also extracts the shared randomness r as part of the secure coin-tossing algorithm. It then plays the role of the honest P_i's and the server while interacting with $\mathcal{A}_1$ until the final step when the majority output label is returned to the parties. Since $\mathbf{Sim}_1$ knows the randomness used

in creating the garbled circuits and the final output $C(\mathbf{x})$, he can compute the output label corresponding to $C(\mathbf{x})$ on his own and send it to $\mathcal{A}_1$. It is easy to see that $\mathcal{A}_1$'s view is identical in the real and ideal execution given a secure coin-tossing algorithm.

- $\mathbf{Sim}_i$'s description and proof of security are essentially identical to that of $\mathbf{Sim}_1$ and hence omitted.

This ends the proof of our first claim.

□

Claim. The protocol securely computes the circuit C in the presence of a malicious server and honest parties.

For this claim, we only need to provide a simulator $\mathbf{Sim}_S$ for the adversary $\mathcal{A}_S$ but unlike the previous claim, $\mathcal{A}_S$ is no longer semi-honest and can actively cheat.

Once again $\mathbf{Sim}_S$ plays the role of the semi-honest parties using fake inputs for each, and following the steps of the protocol. If $\mathcal{A}_S$ returns the wrong majority output, $\mathbf{Sim}_S$ sends abort to the trusted party, simulates the honest parties aborting in order to simulate $\mathcal{A}_S$'s view in the real protocol, and outputs whatever $\mathcal{A}_S$ does. To show indistinguishability of the views note that if $\mathcal{A}_S$ behaves semi-honestly, the argument for our first claim still holds, but if $\mathcal{A}_S$ returns an incorrect majority output label, he will receive an abort message from the honest parties with all but negligible probability in s (in the real model) and receives an abort from $\mathbf{Sim}_S$ (in the ideal execution). The honest parties' views are also the same in both models as they receive an abort from the trusted party in the ideal model when $\mathcal{A}_S$ cheats by sending the wrong majority output, and otherwise receive their correct output.

□

Claim. The protocol securely computes the circuit C in presence of an honest server and an all-but-one set of malicious parties.

Sim needs to simulate $\mathcal{A}$ who corrupts a subset of the parties in the real execution. The simulation works as follows. **Sim** extracts the shared randomness r by playing the role of the honest party P_i during the coin-tossing protocol and uses it to recover the input labels for all possible inputs. Then, when the parties send to the server the labels for their inputs, the simulator can learn their actual inputs and send it to the trusted party in order to receive the output of the computation. It then follows the protocol (sending an abort to the trusted party if it detects cheating by $\mathcal{A}$) until the stage in which it returns the output to $\mathcal{A}$. Here, instead of returning the majority value, it returns the output label ω^b for the output it received from the trusted party (it knows this value since it knows r). Note that although the simulator uses correct garbled circuits with random inputs, the output it returns to the parties is exactly the output of the ideal functionality.

As a result, $\mathcal{A}$'s view is indeed indistinguishable in the two models, and the honest P_i will either abort or learn the correct output with similar distributions in both models.

□

■

3.3 Pipelining with Malicious Garblers

As demonstrated in [29, 42], pipelining the execution of garbled circuits (i.e., garbling and evaluation) can lead to better running times and less memory consumption. In a pipelined execution, each gate is garbled and evaluated "on the fly" and the result of the evaluation is only stored while still needed for future gates. This results in better running times since the evaluator can start the evaluation while the garbler is still garbling the circuit. Furthermore, there is no need to store the entire garbled circuit (which can be several Gigabytes large) at any point during the execution, which leads to better memory usage.

Unfortunately, the pipelining technique is not applicable in the malicious setting and particularly when one applies cut-and-choose techniques. The difficulty with pipelning when one uses cut-and-choose is that for the evaluator to verify/evaluate a garbled gate he needs to know from the beginning of the execution either the circuit's randomness (for verification) or the garbler' s label (for evaluation). However, if the garbler can determine which circuits will be verified and which will be evaluated while the execution is occurring then he can easily cheat, e.g., by sending an invalid garbled circuit for those being evaluated and valid circuits for those being verified.

Our solution to this problem is to let the evaluator learn the required information *obliviously* in the beginning of the protocol, without giving it any information. This is done by executing an OT protocol for each circuit, where the garbler plays the sender in the OT and has two inputs: (1) the randomness used to garble the circuit; and (2) its input labels for the circuit. The evaluator plays the receiver in the OT and decides whether it wants to either learn the randomness and verify the garbled circuit, or learn the garbler's labels and evaluate the circuit. Transforming our protocols to use this technique is straightforward. For example, for our second protocol, we replace the Garbled Circuit Verification and Input Label Transfer stages with the following: for each $\ell \in [s]$, all players share an input key K_ℓ. Before the Circuit Garbling stage, all players send their inputs to all circuits, encrypted under the input keys. Then, the above OT is executed, where P_1's inputs are the pairs (r_ℓ, K_ℓ), and S's inputs are its choices in the cut-and-choose. Now, P_1 sends the garbled circuits and S can either evaluate or verify them on-the-fly. Last, S reveals its choices and they continue the rest of the protocol.

While we use this new pipelining technique in the server-aided setting, the same approach can be used in the standard SFE setting against malicious adversaries.

4 Evaluation

To evaluate the performance of our protocols we designed and built a framework on top of which we can implement any server-aided (and even standard 2SFE) protocol.

4.1 Our Framework

Our framework consists of the following modules:

Circuit generator - Generates a textual representation of a boolean circuit. The circuit can be specified using the FairPlay format [43], or using object-oriented python code similarly to [29, 42].

Crypto library - Provides implementations of hash functions, pseudo-random functions, commitments and symmetric encryption. These are all based on the JAVA SHA-1 and SHA-256 implementations as suggested in [52]. It also includes our implementation of the Peikert, Vaikuntanathan and Waters OT protocol [50] built on top of the JAVA BigInteger package.

Garbled circuit - Given a circuit representation and a PRF key, it garbles the circuit using randomness generated by the PRF. Given input labels and a garbled circuit, it evaluates the circuit and returns the result. Given a garbled circuit and a key to a PRF, it verifies that the given circuit was generated properly using that key. All these functionalities are designed to work in a pipelined fashion (more details below).

Communication library - Handles communication between a set of parties, either peer-to-peer or broadcast.

Since the focus of our work is on efficiency, we do not consider the cost of generating circuit representations. Our design allows us to use any circuit generator, even ones that generate circuits "on-the-fly" such as [29, 42].

Circuit representation. The particular circuit representation we use is similar to FairPlay's format [43] where each gate has a unique identifier and the circuit is represented by specifying the identifiers of the inputs for each gate. As an example, the string `"354 120 380 AND"` represents an AND gate with identifier 354 that uses inputs from gates 120 and 380. We slightly augment the FairPlay format with information that is needed for pipelining the circuits. This is because during pipelining we have to determine in realtime whether a gate is needed to evaluate future gates or not (in which case we can free the memory used for storing its value).

Pipelining. We now discuss how pipelining is implemented. Note that the work of garbler is fairly simple and space-efficient. It goes over the circuit specification and, for each gate, uses the PRF to generate the labels for the input and output wires of the gate. Similar work is done by the evaluator when verifying a circuit.

If, on the other hand, the evaluator evaluates the circuit, pipelining becomes more difficult since it needs to store intermediate values. We can view the pipelining process as a topological ordering of the gates that minimizes the number of *live* gates (i.e., gates that are still needed) in each part of the ordering. Obviously, the optimal ordering can be pre-computed during a pre-processing phase so we assume that the circuit representation is already ordered optimally. During the protocol execution, the evaluator maintains a list of all live gates. When a gate is not needed anymore, the garbler notifies the evaluator that it can free that gate. As a result, the evaluator maintains only the required intermediate values for the rest of the process.

Free XOR. Finally, we mention that we also use the free XOR technique of Kolesnikov and Schneider [35] that is now standard in any garbled circuit implementation. This technique allows us to construct a circuit in such a way that XOR operations are "free" in the sense that they do not require any cryptographic operations.

4.2 Experimental Results

We use two circuits in our experiments: (1) a circuit that given a 128-bit message and a 1408-bit (expanded) key, computes the AES encryption of the message under the key [2]; and (2) a circuit that computes the edit distance of two 50 character strings of 8 bit characters. The size of the AES circuit

[2] AES encryption is currently the standard benchmark for 2SFE implementations. Specifically, we use the circuit of [52] in order to correctly compare it with the results of both [52] and [55]. We remark that any optimization of the AES circuit itself, e.g. as done in [8], could improve performance in a similar way.

is 31512 gates of which 13904 are non-XOR gates. We generated the edit distance circuit according to the suggestions of [29]. The size of that circuit is 254930 gates, 94472 of which are non-XOR gates.

For our experiments we used two Intel Core 2 Duo 3GHz machines with 4GB RAM connected through a switched LAN. The first was used for executing the protocol of S, and the second one for P_1 and the rest of the parties. As suggested in [52] we use a security level of 2^{-40} for the malicious case and of 2^{-4} for the covert case.

In Tables 1 we present a summery of our experiments. These results are with SHA-1 as the underlying hash function. The use of SHA-256 increases the total time by $5\% - 10\%$ and the communication of P_2 (and the other weak parties) by 30% on average in the first protocol (since the size of the commitments is affected). We do not include the running time of the weaker parties since they were significantly smaller than S and $P1$ (e.g., 2 seconds for protocol 2 with AES).

We conclude from the above numbers that: (1) our first protocol requires more communication from the weak parties which makes it suitable mainly for weak devices that have high bandwidth; (2) the complexity of our second protocol is almost independent in the number of parties. The communication and running time of our first protocol, however, increases as the number of parties grows. The main overhead in both protocols is the communication of the garbled circuits.

Comparison to previous implementations. There are many factors that affect performance in practice. For example, size and structure of the circuit, communication latency, the security parameters, etc. However, the important and most objective parameters for evaluating any system are the total running time and the communication time. Thus, we try to compare our results with previous ones with respect to these parameters as much as possible [3]

As we showed in Section 2, any secure 2SFE can be used to achieve the security we require. The question, however, is whether current 2SFE constructions give us better performance. The most efficient 2SFE implementation for semi-honest adversaries is the one of [29], which can securely compute AES in $0.2s$. For the case of a covert adversary, the best result is 60 seconds [52] (in the random oracle model), whereas our first protocol takes around 9 seconds for 2 parties.

As for the more interesting case of 2SFE secure in the malicious model, the implementation of [52] runs in 1114 seconds, whereas the best known result is 192 seconds [55]. We note that the latter excludes communication and that our second protocol is more than four times faster *including communication* (which is of similar size to that of [55]).

A different but relevant class of protocols is, of course, that of secure *multi*-party SFE. In this case, the performance of current implementations is a step behind that of 2SFE implementations. For example, in the work of [13], one 32-bit multiplication takes roughly $9ms$. Since it is based on arithmetic circuits, however, it is not clear how to best compare it to our work. If we multiply its running time by the number of non-XOR gates of the AES circuit, we get around $125s$—which is almost three times slower than our second protocol. This performance, however, is for four parties with at most one malicious party, whereas our protocol with four parties allows up to *three* malicious parties. Moreover, it is shown in [13] that as the number of parties grows, the performance per multiplication gate gets worse (e.g., for seven parties with at most two malicious, it takes $28ms$) whereas the performance of our protocol is almost independent of the number of parties.

The work of [11] considers secure multi-party computation with boolean circuits, but only for semi-honest parties. They do not use the AES benchmark but they show that it takes roughly $3 - 8$ seconds for three parties to evaluate a circuit with 5500 AND gates. This means that in order to be faster than our second protocol, a semi-honest-to-malicious transformation would need to have an multiplicative

[3] We note that some works try to compare the different steps of the protocols, but we find such comparisons to be of little value.

overhead smaller than 6. Recall that in the two-party case, the efficiency ratio between semi-honest and malicious security is more than several hundreds.

Side-channels in pipelined execution. We note that during our experiments, we experienced different timings for the processing of the circuits that were evaluated and those that were verified. This seems inevitable since the receiver works harder in case he checks a garbled circuit. Indeed, simple solutions like restricting the receiver to work in constant time could work, at the cost of efficiency. However, clever techniques that parallelize the work on several circuits could be more efficient. We leave this direction for future work.

	AES				Edit distance	
	Protocol 1 2 parties	Protocol 1 4 parties	Protocol 2 2 parties	Protocol 2 4 parties	Protocol 1 2 parties	Protocol 2 2 parties
Total Time	9.12	14.8	45	46	33.5	240
Communication Time	6.5	9.5	32	32	26	185
P1 / S Communication	27777KB	27777KB	216749KB	216749KB	165918KB	1296319KB
P2 Communication	2443KB	5539KB	33KB	33KB	862KB	2KB

Table 1: Experimental results. Total time is the sum of communication time and computation time (in seconds). P_1 / S communication is the communication size of the party who communicates the most (either P_1 or S). P_2 communication is the communication size of any one of the weaker players.

Acknowledgments

We would like to thank Benny Pinkas and Nigel P. Smart for providing us the AES circuit from [52], and Peeter Laud for his valuable comments.

References

[1] G. Asharov, A. Jain, A. Lopez-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *EUROCRYPT*, 2012.

[2] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC*, 2007.

[3] B. Barak and O. Goldreich. Universal arguments and their applications. In *CCC*, 2002.

[4] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *CCS*, 2008.

[5] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, 2008.

[6] P. Bogetoft, D. Christensen, I. Damgard, M. Geisler, T. Jakobsen, M. Krøigaard, J. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In *FC*, 2009.

[7] P. Bogetoft, I. Damgard, T. P. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. A practical implementation of secure auctions based on multiparty integer computation. In *FC*, 2006.

[8] J. Boyar and R. Peralta. A small depth-16 circuit for the aes s-box. In *Information Security and Privacy Research*, 2012.

[9] R. Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology*, 2000.

[10] D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *STOC*, 1988.

[11] S. G. Choi, K. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In *CT-RSA*, 2012.

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

[13] Ivan Damgaard, Martin Geisler, Mikkel Kroigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In *PKC*, 2009.

[14] I. Damgard, S. Faust, and C. Hazay. Secure two-party computation with low communication. In *TCC*, 2012.

[15] I. Damgard, M. Geisler, M. Krøigaard, and J.-B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *PKC*, 2009.

[16] I. Damgard and Y. Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In *CRYPTO*, 2005.

[17] I. Damgard, Y. Ishai, M. Krøigaard, J.-B. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *CRYPTO*, 2008.

[18] U. Feige, J. Killian, and M. Naor. A minimal model for secure computation (extended abstract). In *STOC*, 1994.

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

[20] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *Advances in Cryptology - CRYPTO '10*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer-Verlag, 2010.

[21] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[22] O. Goldreich. *Foundations of Cryptography – Volume 2.* Cambridge University Press, 2004.

[23] O. Goldreich. *Foundations of Cryptography – Volume 1.* Cambridge University Press, 2006.

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

[25] D. Gordon, J. Katz, V. Kolesnikov, T. Malkin, M. Raykova, and Y. Vahlis. Secure computation with sublinear amortized work. Technical Report 2011/482, IACR ePrint Cryptography Archive, 2011.

[26] S. Gordon and J. Katz. Partial fairness in secure two-party computation. *EUROCRYPT*, 2010.

[27] S. D. Gordon, C. Hazay, J. Katz, and Y. Lindell. Complete fairness in secure two-party computation. *Journal of the ACM (JACM)*, 58(6):24, 2011.

[28] W. Henecka, S. Kogl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS*, 2010.

[29] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[30] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, 2003.

[31] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Garbled circuits for leakage-resilience: hardware implementation and evaluation of one-time programs. In *CHES*, 2010.

[32] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party comptuation. Technical Report 2011/272, IACR ePrint Cryptography Archive, 2011.

[33] J. Katz, R. Ostrovsky, and A. Smith. Round efficiency of multi-party computation with a dishonest majority. In *EUROCRYPT*, 2003.

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

[35] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In *ICALP*, 2008.

[36] B. Kreuter, a. shelat, and C.-H. Shen. Towards billion-gate secure computation with malicious adversaries. Technical Report 2012/179, IACR ePrint Cryptography Archive, 2012.

[37] Y. Lindell. Parallel coin-tossing and constant-round secure two-party computation. In *CRYPTO*, 2001.

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

[39] Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 2009.

[40] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *TCC*, 2011.

[41] Y. Lindell, B. Pinkas, and N. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *SCN*, 2008.

[42] L. Malka. Vmcrypt: modular software architecture for scalable secure computation. In *CCS*, 2011.

[43] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, 2004.

[44] S. Micali and P. Rogaway. Secure computation (abstract). In *CRYPTO*, 1992.

[45] P. Mohassel and M. Franklin. Efficiency tradeoffs for malicious two-party computation. In *PKC*, 2006.

[46] M. Naor and K. Nissim. Communication preserving protocols for secure function evaluation. In *STOC*, 2001.

[47] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *STOC*, 1999.

[48] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SODA*, 2001.

[49] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *EC*, 1999.

[50] C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO*, Berlin, Heidelberg, 2008.

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

[52] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure two-party computation is practical. In *ASIACRYPT*, 2009.

[53] M. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Aiken Computation Lab, Harvard University, 1981.

[54] A. Shamir. How to share a secret. *Commun. ACM*, November 1979.

[55] A. Shelat and C. H. Shen. Two-output secure computation with malicious adversaries. In *EUROCRYPT*, 2011.

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

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

[58] A. Yao. How to generate and exchange secrets. In *FOCS*, 1986.