Transforming Secure Comparison Protocol from Passive to Active Adversary Model

Wei Jiang

Department of Electrical Engineering and Computer Science
The University of Missouri - Columbia
wjiang@missouri.edu

Abstract

Secure comparison (SC) is an essential primitive in Secure Multiparty Computation (SMC) and a fundamental building block in Privacy-Preserving Data Analytics. Although secure comparison has been studied since the introduction of SMC in the early 80s and many protocols have been proposed, there is still room for improvement, especially providing security against malicious adversaries who form the majority among the participating parties. It is not hard to develop an SC protocol secure against malicious majority based on the current state of the art SPDZ framework. SPDZ is design to work for arbitrary polynomially-bounded functionalities, and it may not provide the most efficient SMC implementation for a specific task, such as SC. In this paper, we propose a novel compiler that is specifically designed to convert most existing SC protocols with semi-honest security into the ones secure against the malicious majority. This compiler provides a flexible and efficient way to achieve both covert and active security for passively secure SC protocols.

1 Introduction

Comparison serves as one of the most fundamental operators in various data analytics. When the data considered under these applications contain sensitive information and are from multiple sources, privacy-preserving data analytics (PPDA) protocols may have to be adopted to protect the data and the outcomes. Secure Multiparty Computation (SMC) primitives sever as essential building blocks for developing many existing PPDA protocols. Among the SMC primitives, secure comparison (SC) enjoys widespread adoption.

Although, SMC techniques provide very strong guarantee on personal privacy and data security, they are computationally expensive. For the last three decades, significant efforts have been devoted into developing efficient SMC primitives including SC. Current SC implementations can be classified into several categories based on the underlying building blocks, such as garbed circuits [32,45], homomorphic encryption [19,39], secret sharing [17,38,40,41,43], and the SPDZ framework [3, 20, 22, 36]. Additionally, the existing SC protocols can also be classified based on their security guarantees, such as semi-honest, covert and malicious [1, 2, 27].

If a protocol is secure under the semi-honest assumption, the participating parties are expected to follow the execution requirement of the protocol but may use what they see during the execution to compute more than they need to know. If a protocol is secure under the malicious assumption, the participating parties can diverge arbitrarily from the normal execution of the protocol. The semi-honest adversarial model often leads to more efficient privacy-preserving protocols than the malicious model, but the malicious model is less restrictive and thus more realistic. To take advantage of both, the covert model [1, 2], a sub-class of the malicious model, allows the computation to diverge arbitrarily (as with the malicious), but provide certain guarantees of detectability of such behaviors. As a result, protocols satisfy covert security are more efficient than the ones that guarantee full malicious security.

In this paper, we use semi-honest, covert, or malicious security to mean the security under the semi-honest, covert, or malicious model respectively. What kind of security can be achieved also depends on the number of malicious parties among the participating entities. When the majority (e.g., $n-1$ out of n) of the participating parties are malicious, to our knowledge, full malicious security cannot be achieved at least for SC protocols. The best can be done is to detect if any party behaved maliciously during protocol execution.

These SC protocols [17, 38, 40, 41] are secure under the malicious model. However, their security is only guaranteed when the number of malicious parties is less than half of the total parties involved in the protocol execution. When the majority of parties are malicious (or malicious majority), the current state of the art implementation of an SC to achieve malicious security is to adopt the SPDZ framework [3, 22, 36] that combines additive secret sharing and fully homomorphic encryption [11]. The framework utilizes zero-knowledge (ZK) proofs and triple sacrifice techniques [4, 35] to detect malicious behaviors. ZK proofs are computationally expensive, and SPDZ is a general framework to implement maliciously secure SMC protocols. Thus, it may not be the best tool to design very efficient and maliciously secure SC protocols.

1.1 Our Contribution

The goal of this paper is to propose a novel technique, termed as *randomized replication*, to develop a compiler that transforms semi-honestly secure SC protocols to be secure against malicious majority. More specifically, we consider the client-server computing model where clients outsource their data and analytics tasks to two or more independent servers. Most existing SMC solutions are applicable in the model. Our proposed compiler is also generic in that a newly developed and more efficient secure comparison protocol can be used without changing most of its code or structure.

Since the clients are not involved in protocol execution, the servers are commonly referred to as the participating parties. As a result, an SC protocol under the client-server model may be formulated as:

$$\mathrm{SC}\left(\left\langle P_i, [a]^{P_i}, [b]^{P_i}\right\rangle\right) \rightarrow \left\langle P_i, [\tau]^{P_i}\right\rangle \tag{1}$$

where a and b are actual values being compared, $[a]^{P_i}$ and $[b]^{P_i}$ are secret shares of a and b that are possessed by party P_i, and i varies from 1 to n. The comparison result is represented by τ, secretly shared among the n parties. In this work, we propose a novel compiler that transforms an SC protocol with semi-honest security into a maliciously secure SC protocol that is more efficient than the current state of the art SPDZ based solutions [3, 22, 36]. Our compiler overcomes the high cost of the SPDZ protocols using a simple yet effective treatment. Our contributions are as follows:

- We present a novel compiler that executes an semi-honest SC protocol κ times with randomized and replicated inputs plus end protocol verification, where κ is a user-chosen statistical security parameter.
- The κ parameter in our compiler is independent of the underling fully homomorphic encryption (FHE) scheme. This leads to a small polynomial dimension N and a short ciphertext.
- Our protocol achieve security in both covert and malicious models only by adjusting the actual value of κ.

1.2 Security Guarantee and Threat Model

Let n denote the number of parties/servers, and up to $n-1$ parties can be malicious. The assumption of computing power of these parties depends on the actual design and implementation of the SC protocols being transformed using the proposed compiler. For example, if an SC protocol assumes the parties are computationally unbounded, then the same assumption holds under our compiler. The security of an SMC protocol may have several criteria:

- Privacy: the private input data of an honest party is not disclosed to the other parties during protocol execution.
- Correctness: in presence of malicious behaviors, the honest parties can still receive the correct output.
- Fairness: either every party receives the correct output or no parties receive the correct output.
- Detectability: any malicious behaviors can be detected.

Any SMC protocols have to guarantee privacy, but the other properties may or may not be achieved depending on the number of malicious parties. For example, correctness may be achievable if the number of malicious parties is less than $\frac{n}{3}$ with Shamir's secret sharing scheme.

Under the malicious majority setting, in addition to privacy, the existing SMC protocols only guarantee detectability of malicious behaviors. Malicious behaviors generally mean collusion among the parties and not following the protocol, all of which are equivalent to changing or modifying the shares. For instance, suppose a value $v = v_1 + v_2 + v_3 \bmod p$ is secretly shared among three parties: P_1, P_2 and P_3. Each P_i has the share v_i. Suppose v is a private input of an SMC protocol, and P_1 and P_2 are malicious. Then any malicious behaviors of P_1 and P_2 are equivalent to using v_1' and v_2' as their shares during protocol execution where v_1' and v_2' may or may not be the same as v_1 and v_2. Therefore, except for prematurely aborting the protocol, detecting malicious behaviors actually means the protocol can detect or verify if the shares have been modified.

In summary, our proposed solutions provide detectibility when the number of colluding and malicious parties is bounded by $n - 1$, and the participating parties have either limited or unlimited computing power depending on the implementation of the underlying SC protocols.

1.3 Organization

The rest of the paper is organized as follows: Section 2 presents the background and the work most relevant to ours. Section 3 discusses the underlying tools required to build our protocol. Section 4 proposes a novel compiler that can transform most SC protocols to satisfy security against malicious majority. Section 5 concludes the paper with future research directions.

2 Related Work

Secure Multiparty Computation (SMC) was first introduced by Yao's Millionaire problem for which a provably secure solution was developed [44,45]. This was extended to multiparty computations by Goldreich et al. [28]. SMC can be categorized as either computational [14,28,45] or information theoretic [9,13]. In the computational model, the adversary is assumed to be bounded by polynomial-time. In the information theoretic model, the adversary is assumed to be unbounded. Much work exists to address various aspects (e.g., complexity, adversarial behaviors, the number of corrupted parties) of SMC.

There are three main types of adversaries related to the SMC definitions: semi-honest, covert, and malicious [1,2,27]. If a protocol is secure under the semi-honest assumption, the participating parties are expected to follow the execution requirement of the protocol but may use what they see during the execution to compute more than they need to know. If a protocol is secure under the malicious assumption, the participating parties can diverge arbitrarily from the normal execution of the protocol. The semi-honest adversarial model often leads to more efficient privacy-preserving protocols than the malicious model, but the malicious model is less restrictive and thus more realistic. To take advantage of both models, the covert model [1,2] allows the computation to diverge arbitrarily (as with the malicious), but provide certain guarantees of detectability of such behaviors.

2.1 SMC in the Malicious Model with Honest Majority

A number of SMC solutions have been proposed to address the security issues under the malicious majority adversary model. Here we summarize a few representative work in this area, where the functionalities are represented as arithmetic circuits. A verifiable secret sharing scheme is proposed in [25] which uses Shamir's secret sharing and homomorphic commitments based on the discrete log assumption. The commitments are updated along the computations, and any malicious changes to the computation will lead to inconsistent commitments. On the other hand, the commitment scheme is computationally expensive. Combining dispute control [6] and utilizing a designed party for intermediate computation [31], more efficient solutions were introduced in [21] that has linear complexity based on the circuit size, and its verification technique has error probability negligible in terms of pre-defined security parameter. To remove this error probability, [7] proposed to use hyper-invertible matrices to perform batched correctness check of shares that leads to a perfectly secure solution assuming the number of malicious parties is less than $\frac{n}{3}$.

In [29], a technique of 4-consistent tuples of shares was introduced to improve the communication complexity given in [7] by removing the quadratic terms in the multiplicative depth of the circuit. To ensure perfect security in presence of malicious parties, all these solutions assume the number of malicious parties is less than $\frac{n}{3}$. Based on the solution given in [10], the work [30] proposed a solution with linear complexity and less than $\frac{n}{2}$ parties can be malicious. More efficient solutions exist, but they can only detect whether or not the parties have followed the protocol.

In [37], a framework was introduced to allow computations performed by using a semi-honest protocol along with an efficient verification steps to check the correctness of a set of Beaver triples [5]. In [15], a circuit randomization technique was proposed to verify the consistencies between two executions: one on the original circuit and one on a randomized circuit by multiplying the inputs with a random value. Both works [15,37] assume that the multiplication protocol is secure up to additive attacks [23,24],[1] and the number of malicious parties is less than $\frac{n}{2}$.

2.2 SMC against Malicious Majority

For malicious majority (up to $n-1$), designing efficient SMC protocol gets more and more challenging. The well-known SPDZ framework was proposed in [22] and later improved in [20] with cut-and-choose techniques instead of ZK proofs [16] to detect malicious behaviors. The framework utilizes fully homomorphic encryption scheme [11,26] to efficiently produce multiplication triples. MASCOT [35] adopts OT extensions [33] to produce the triples more efficiently than the earlier implementations of the SPDZ framework. Due to more efficient

[1] Additive attack means that an adversary can add any chosen value to the output of a secure multiplication.

ZK proofs and other advancement, recent SPDZ based solutions offer the best performance [3, 8, 34, 36] in this adversary setting.

2.3 Secure Comparison

A number of methods have been proposed for this functionality, but we only discuss the designs that can be incorporated into the SPDZ framework to produce an SC protocol secure against malicious majority. These designs were started by the constant round SC protocol of Damgård et al [18]. In this design, secretly shared values must first be bit decomposed among the parties involved in the computation. Alternatively, the values may exist as bitwise shares initially. This means the protocol takes as input bit decomposed shares of the private values to be compared. If this procedure is necessary, though expensive, it is potentially beneficial when other bit-wise operations may be seen as advantageous. The cost incurred in this scheme for bit decomposition may be amortized somewhat across all those sub-protocols which require it. Though there is a fairly high computational complexity and communication cost, this important result demonstrates constant rounds secure comparison is indeed possible and well within feasibility.

The latter strategy, that of exploiting properties of finite field arithmetic, seeks to affect a comparison through intermediate comparisons and some logic to bring the meaning of these intermediate comparisons together to form the desired solution. This method was introduced in the work of Nishide and Ohta [38]. Other optimizations have since come which cut back on the complexity and the number of intermediate calculations necessary based on some restrictions to the domain of values [40, 41]. The proposed transformation technique works on all existing secure comparison protocols.

3 Preliminaries and Definitions

This section provides background on the secret sharing scheme used to construct the proposed protocols, commonly used notations and security definitions.

3.1 Conventions and Notations

The following notations are commonly used in the literature and for the rest of the paper:

- $P_1, \ldots, P_n$: n parties or servers who collaboratively and securely perform the required computations.
- $\mathbb{Z}_p$: a prime domain $\{0, \ldots, p-1\}$ where p is a prime and $|p| = l$, the number of bits needed to represent p.
- $[x]$: a value x is secretly shared among the n parties. The shares are drawn from $\mathbb{Z}_p$. The domain of x is bounded by p. Whenever the context is clear, we drop p from the notation, i.e., $[x]$.

- $[x]^{P_j}$ (or $[x]^{P_j}$): the secret share of x belongs to party P_j. Thus, $[x]$ is a set of shares denoted by $[x]^{P_1}, \ldots, [x]^{P_n}$.

The superscript in $[x]^{P_j}$ may also be dropped for succinctness. For example, the expression below represents local computations performed by P_j based on its own shares.

- P_j: $[x]^{P_j} \leftarrow \sum_{j=1}^{n} [x^j]^{P_j}$

It produces P_j's secret share of x by summing each secret share of $[x^j]^{P_j}$. To simplify the notations, we often adopt the following expression instead:

- P_j: $[x] \leftarrow \sum_{j=1}^{n} [x^j]$

Moreover, the term "secret share" (or "secretly shared") is interchangeable with "share" (or "shared").

3.2 Secret Sharing and its Functionalities

We require that any secret sharing scheme to be used have the ability to perform the following operations, and the ones required communications among the parties are denoted as ideal functionalities with symbol $\mathcal{F}$. As stated previously, we assume that the adversary $\mathcal{A}$ is computationally bounded and control at most $n-1$ parties who remain the same throughout the protocol execution. We only need to guarantee privacy.

- The adversary does not learn any information about the private input of an honest party.

Since we do not need to guarantee the computation correctness, the implementations of these functionalities are highly efficient. Note that detecting malicious behaviors is achieved through our proposed compiler instead of at the sub-protocol level which leads to a very efficient SC implementation against the malicious majority.

- $\mathcal{F}_{\text{share}}(x)$: given a particular value $x \in \mathbb{Z}_p$, a dealer can generate shares $[x]^{P_1}, \ldots, [x]^{P_n} \in \mathbb{Z}_p$ of x. Each party P_j has share $[x]^{P_j}$. This must be done in a way that they can be uniquely recombined in a method applicable to the scheme to reconstruct the original value.

- $\mathcal{F}_{\text{open}}([x])$: all n shares $[x]^{P_j}$ are needed to reconstruct x.

- $\mathcal{F}_{\text{mult}}([x], [y])$: given two secretly shared values x and y, it returns secret shares of xy. Specifically, the functionality returns $[xy]^{P_j}$ to party P_j.

- $\mathcal{F}_{\text{mult}_2}(\langle P_i, \alpha\rangle, \langle P_j, \beta\rangle)$: a two-party functionality that allows P_i with private input α and P_j with private input β to derive $[\alpha\beta]^{P_i}$ and $[\alpha\beta]^{P_j}$.

- Local operations:

- Addition with a public constant: given shares $[x]$, and a public constant c, execute the necessary operations to calculate $[c+x]$.
- Addition: given two shared values $[x]$ and $[y]$, calculate the shares of the sum of the original values $[x+y]$.
- Multiplication by a public constant: given shares $[x]$, and a public constant c, execute the necessary operations to calculate $[cx]$.

We adopt additive secret sharing that satisfies the above requirements. Here we briefly discuss how to implement each functionality and local operations.

- $\mathcal{F}_{\text{share}}(x)$: to share x, a dealer (or any P_j) randomly selects $s_1, \ldots, s_{n-1} \in \mathbb{Z}_p$, and computes:

  - $[x]^{P_j} \leftarrow s_j$, for $1 \le j \le n-1$.
  - $[x]^{P_n} \leftarrow \left(x - \sum_{j=1}^{n-1} s_j\right) \bmod p$.

  The dealer sends $[x]^{P_j}$ to party P_j, for $1 \le j \le n$.

- $\mathcal{F}_{\text{open}}([x])$: P_j broadcasts $[x]^{P_j}$ to the other parties. Then each party locally compute $x \leftarrow \sum_{j=1}^{n} [x]^{P_j} \bmod p$.

- $\mathcal{F}_{\text{mult}}([x],[y])$: This functionality is implemented using $\mathcal{F}_{\text{mult}_2}$ according to the following observation.

$$\begin{aligned} xy &= \left([x]^{P_1} + \cdots + [x]^{P_n}\right)\left([y]^{P_1} + \cdots + [y]^{P_n}\right) \\ &= \sum_{i=1}^{n} [x]^{P_i}[y]^{P_i} + \sum_{i=1}^{n} \sum_{j=1 \land i \ne j}^{n} [x]^{P_i}[y]^{P_j} \end{aligned}$$

  The first summation is computed locally by each party, and the second summation utilizes the $\mathcal{F}_{\text{mult2}}$ functionality to produce secret shares of multiplication between each $\langle [x]^{P_i}, [y]^{P_j} \rangle$ pair. Then the share of $[xy]$ for each party can be derived by summing all its local shares:

$$[xy]^{P_i} \leftarrow [x]^{P_i}[y]^{P_i} + \sum_{j=1 \land j \ne i}^{n} \mathcal{F}_{\text{mult2}}\left([x]^{P_i}, [y]^{P_j}\right)$$

- $\mathcal{F}_{\text{mult}_2}(\langle P_i, \alpha \rangle, \langle P_j, \beta \rangle)$: the implementation of $\mathcal{F}_{\text{mult}_2}$ uses homomorphic encryption (FHE) provided by the SEAL library [42]. The suitable FHE parameters are chosen by P_i according to the plaintext domain and the required security level.

- Local operations:

  - Addition with a public constant: given shares $[x]$, and a public constant c, let P_1 be a designed party. Then $[c+x]^{P_1} \leftarrow c + [x]^{P_1}$ and $[c+x]^{P_j} \leftarrow [x]^{P_j}$ for $2 \le j \le n$.

– Addition: given two shared values $[x]$ and $[y]$, each party sets $[x+y]^{P_j} \leftarrow [x]^{P_j} + [y]^{P_j}$.

– Multiplication by a public constant: given $[x]$, and a public constant c, each party sets $[cx]^{P_j} \leftarrow c[x]^{P_j}$.

3.3 Generating Secretly Shared Random Values

The proposed protocols require the parties to secretly share a random bit or a random value and compute the inverse of a secretly shared value from $\mathbb{Z}_p$. We define these functionalities below and their implementations.

- $\mathcal{F}_{\text{rand}}(p)$: generating a random value r in $\mathbb{Z}_p$ and secretly sharing it among the n parties. At the end, P_j holds share $[r]^{P_j}$, and no parties know r.
- $\mathcal{F}_{\text{rand}_b}(p)$: generating a random bit $\tau \in \{0,1\}$ and secretly sharing it among the n parties. At the end, P_j holds share $[\tau]^{P_j}$, and no parties know τ.
- $\mathcal{F}_{\text{invert}}([r])$: generating shares of $\left[r^{-1}\right]$, where r^{-1} is the multiplicative inverse of r in $\mathbb{Z}_p$. At the end, P_j holds share $\left[r^{-1}\right]^{P_j}$, and no parties know r^{-1}.

In these functionalities, p also defines the domain for the random shares. Existing protocols for these functionalities can be found in [17].

4 Compiler for Transforming Secure Comparison against the Malicious Majority

In the previous protocols, a malicious party can modify the shares to produce invalid comparison results. By doing so, the malicious party would have a very good chance of flipping the comparison outcome. The attack success rate is about $\frac{1}{2}$ analyzed below.

Except for aborting the protocol, a malicious behavior during protocol execution is equivalent to share modification. Therefore, we merely need to estimate the probability that by modifying the shares, how likely the comparison result of SC will change. The following attack is feasible and may flip the result:

- The adversary $\mathcal{A}$ modifies the shares $[a]^{\mathcal{A}}$ and $[b]^{\mathcal{A}}$ during an execution of $\mathcal{F}_{\text{sc}}$ or an SC protocol.

Let E be the event of flipping the comparison result by modifying the shares, and the probability of E can be estimated as follows assuming $a \geq b$ and $a < b$

are equally likely to happen in practice:

$$\begin{aligned}\mathcal{P}(E) &= \mathcal{P}(a \geq b)\mathcal{P}(E|a \geq b) + \mathcal{P}(a < b)\mathcal{P}(E|a < b) \\ &= \frac{1}{2}\mathcal{P}(E|a \geq b) + \frac{1}{2}\mathcal{P}(E|a < b) \\ &= \frac{1}{2}\left(\mathcal{P}(b > a) + \mathcal{P}(b \leq a)\right) \\ &= \frac{1}{2}\end{aligned}$$

Such an attack can be easily carried out because the malicious party knows exactly which share to modify. In what follows, we will propose novel strategies to reduce the attack success rate to a negligible one.

Let κ be a statistical security parameter, and our goal is to detect malicious behaviors with probability bounded by $1 - \frac{1}{2^\kappa}$. To achieve this, the key idea in our design is for the participating parties to execute κ independent copies of the SC protocol with randomized input. By randomization, we mean the input shares $\langle [a], [b] \rangle$ are randomly permuted so that the malicious parties will not be able to consistently alter the shares across all κ copies. The following steps are needed to transform any semi-honest secure SC protocols into a secure one under the malicious model.

- Input commitment: The clients provide their inputs to the servers along with commitments to prevent input modification by the malicious servers.
- Randomized input replications: After the servers receive the shares, they randomize the ordering of the input to produce κ copies of the input pairs for subsequent κ independent SC computations.
- Output verification: Checking if all κ copies produce the same outputs.

4.1 Input Commitment

For the rest of this section, we use a triple $[a, \theta_a, \delta_a]$ to represent shares of the input value a, where δ_a is randomly chosen from $\mathbb{Z}_p^+$, and $\theta_a = a \cdot \delta_a$. The value θ_a serves as a message authentication code (MAC) for a. Such code is also used in [35, 36]. To verify if the shares of a have been modified or not, the parties can perform the following verification steps:

- Collaboratively generate a random secretly shared value r from $\mathbb{Z}_p^+$. As discussed in [17], this can be done by randomly generate $[r]$ and $[s]$ from $\mathbb{Z}_p$. $[t] \leftarrow \mathcal{F}_{\text{mult}}([r], [s])$ and $\mathcal{F}_{\text{open}}([t])$. If $t = 0$, repeat these steps; otherwise, return $[r]$.
- Compute $[\omega] \leftarrow [r]([\theta_a] - [a][\delta_a])$.
- $\mathcal{F}_{\text{open}}([\omega])$ and examine:
  - If $\omega = 0$, verification passed.

– If $\omega \neq 0$, verification failed.

As long as one party follows the steps, any modifications to the shares can be detected with probability $1 - \frac{1}{p}$. The goal of this verification is to make sure that when inputs are replicated, any malicious changes to the shares can be detected before executing the SC protocol.

4.2 Input Randomization and Replication

Input commitment only guarantees the inputs are valid; however, it cannot be used to verify if the parties followed the the prescribed steps during protocol execution. To be able to verify if the parties followed the protocol, the parties execute SC κ times in parallel with randomized inputs. For example, we flip a coin κ times, the outcome of each coin flip is denoted by t_i for $1 \leq i \leq \kappa$. For succinctness, we use $[[a]]$ to represent $\langle [a], [\theta_a], [\delta_a] \rangle$, and adopt $\langle [[x]], [[y]], [t_i] \rangle$ or $\langle [[x]], [[y]] \rangle_{t_i}$ to represent the randomized input for the i-th SC execution based on t_i:

- $\langle [[x]], [[y]] \rangle_{t_i} = \langle [[a]], [[b]] \rangle$, if $t_i = 0$
- $\langle [[x]], [[y]] \rangle_{t_i} = \langle [[b]], [[a]] \rangle$, if $t_i = 1$

The randomization of the input shares has to be performed in an oblivious way so that the parties do not know which input shares are actually swapped. To achieve this, the parties perform the steps given in Protocol 1:

- Step 1: the parties generate a shared random bit $[t_i]$ for randomizing each input pair, and it will also be used to de-randomize the output of the i-th SC execution.
- Step 2: from $[t_i]$, the parties generate a permutation matrix $[M_{t_i}]$ for each input pair $\langle [[a]], [[b]] \rangle$. When $t_i = 0$, M_t is the 2-by-2 identity matrix. When $t_i = 1$, M_{t_i} is a transpose of the 2-by-2 identity matrix.
- Step 3: randomize each input pair by securely multiplying $\langle [[a]], [[b]] \rangle$ with $[M_{t_i}]$. Note that the secure matrix multiplication is applied to each pair of components of $[[a]]$ and $[[b]]$, i.e., $\langle [a], [b] \rangle$, $\langle [\theta_a], [\theta_b] \rangle$ and $\langle [\delta_a], [\delta_b] \rangle$. At the end, the protocol produces $\langle [[x]], [[y]], [t_i] \rangle$.

Protocol 1 Input_Rand($[[a]], [[b]]$) $\rightarrow \langle [[x]], [[y]], [t_i] \rangle$

```
Require: $p$ is a prime defining the share domain.
1: $[t_i] \leftarrow \mathcal{F}_{\text{rand}_b}(p)$
2: $[M_{t_i}] \leftarrow \begin{Bmatrix} [1-t_i] & [t_i] \\ [t_i] & [1-t_i] \end{Bmatrix}$
3: $\langle [[x]], [[y]] \rangle_{t_i} \leftarrow \langle [[a]], [[b]] \rangle \times [M_{t_i}]$
4: return $\langle [[x]], [[y]], [t_i] \rangle$
```

At the step 3 of Protocol 1, to derive the shares of $[1 - t_i]$, a designated party, say P_1, sets its share $[1 - t_i]^{P_1} \leftarrow 1 - [t_i]^{P_1}$, and the other parties set their shares to $[1 - t_i]^{P_j} \leftarrow p - [t_i]^{P_j}$. The computations for the rest of the protocol can be carried out normally with standard secure additions and multiplications. The same set of permutation matrices can be used to permute the inputs for a number of SCs on different inputs as long as the comparison results are not leaked until the end of all SC executions.

Functionality 2 $\mathcal{F}_{\mathrm{sc}_m}\left(\left\langle [[a]]^{\mathcal{A}}, [[b]]^{\mathcal{A}} \right\rangle, \left\langle [[a]]^{\bar{\mathcal{A}}}, [[b]]^{\bar{\mathcal{A}}} \right\rangle\right) \rightarrow [[\tau]]^{\mathcal{A}}, [[\tau]]^{\bar{\mathcal{A}}}$

```
Require: $\left\langle [[a]]^{\mathcal{A}}, [[b]]^{\mathcal{A}} \right\rangle$ indicates the set of authenticated input shares controlled by the adversary $\mathcal{A}$, and $\left\langle [[a]]^{\bar{\mathcal{A}}}, [[b]]^{\bar{\mathcal{A}}} \right\rangle$ refers to the set of authenticated shares from honest parties. $\kappa$ is the security parameter.
1: $\mathcal{F}_{\mathrm{sc}_m}$ receives $\left\langle [[a]]^{\bar{\mathcal{A}}}, [[b]]^{\bar{\mathcal{A}}} \right\rangle$ from the honest parties, denoted by $P_{\bar{\mathcal{A}}}$. It also receives $\left\langle [[a]]^{\mathcal{A}}, [[b]]^{\mathcal{A}} \right\rangle$ from $\mathcal{A}$.
2: From these shares, $\mathcal{F}_{\mathrm{sc}}$ derives $[a', \theta_{a'}, \delta_{a'}]$ and $[b', \theta_{b'}, \delta_{b'}]$:
    (a) If $\theta_{a'} \neq a' \cdot \delta_{a'}$ or $\theta_{b'} \neq b' \cdot \delta_{b'}$, send abort message to all parties.
3: Set $\tau = 0$ if $a' \geq b'$; otherwise, set $\tau = 1$.
4: Construct authenticated shares of $[[\tau]]^{\mathcal{A}}$ and $[[\tau]]^{\bar{\mathcal{A}}}$ according to the security parameter $\kappa$, and send $[[\tau]]^{\mathcal{A}}$ to $\mathcal{A}$.
5: $\mathcal{F}_{\mathrm{sc}_m}$ waits for reply from $\mathcal{A}$:
    (a) If the reply is abort, send abort to $P_{\bar{\mathcal{A}}}$.
    (b) If the reply is continue, send $[[\tau]]^{\bar{\mathcal{A}}}$ to $P_{\bar{\mathcal{A}}}$.
```

Protocol 3 Verify $([[a]])$

```
Require: $[[a]] \equiv \langle [a], [\theta_a], [\delta_a] \rangle$
1: $[t] \leftarrow \mathcal{F}_{\mathrm{mult}}([a], [\delta_a])$
2: $[s] \leftarrow [t] - [\theta_a]$
3: Return $\mathcal{F}_{\mathrm{verify\_zero}}([s])$
```

4.2.1 Randomizing Inputs with Multiple Components

To permute a pair of random triples, we could apply the same permutation matrix on the corresponding components of each triple. Alternatively, we could first pact the three components of a triple into a bigger share and generate a permutation matrix in a larger field to contain the bigger share. After multiplying the pair of the bigger shares with the permutation matrix, we can unpact the bigger shares to produce a permuted pair of triples.

Protocol 4 Verify_Zero $([s])$

```
1: $[r_1] \leftarrow \mathcal{F}_{\text{rand}}(p)$ and $[r_2] \leftarrow \mathcal{F}_{\text{rand}}(p)$
2: $[t] \leftarrow \mathcal{F}_{\text{mult}}([r_1],[r_2])$
3: $t \leftarrow \mathcal{F}_{\text{open}_m}([t])$
4: Abort the protocol if $\mathcal{F}_{\text{open}_m}([t])$ aborts.
5: If $t = 0$, repeat the previous steps.
6: $[t'] \leftarrow \mathcal{F}_{\text{mult}}([r_1],[s])$
7: $t \leftarrow \mathcal{F}_{\text{open}_m}([t'])$
8: Abort the protocol if $\mathcal{F}_{\text{open}_m}([t])$ aborts.
9: If $t' = 0$, return true; otherwise, return false.
```

4.3 Output Verification

The modified comparison will be performed κ times. After that we need to verify the consistency of these results. Let $[z_1], \ldots, [z_\kappa]$ be the results running the protocol κ times on randomized inputs. Before verifying the consistency of the results, we need to de-randomize them based on the t_i values used to randomize the input shares. However, we cannot simply de-randomize the results since the probability that the adversary alters the de-randomized results without being detected is no longer negligible. For instance, if the results are 0, then the adversary could add one to the share it controls across all κ results. This causes the comparison result flipped, and it is not guaranteed to detect such malicious behaviors. As a solution, the parties need to authenticate the shares of z_i and t_i to produce $[[z_i]]$ and $[[t_i]]$ before de-randomization.

4.3.1 Share Authentication and Result De-randomization

The parties commit or authenticate their shares of the results before de-randomization. Let $[[z_1]], \ldots, [[z_\kappa]]$ be the authenticated shares of $z_1, \ldots, z_\kappa$, and $\tau_1, \ldots, \tau_\kappa$ be the de-randomized results. The parties can derive $[[\tau_i]]$ from $[[z_i]]$ and $[[t_i]]$, and all these authenticated shares use a global and secretly shared MAC key δ (see Section 4.4 for the details). To verify if the τ_i values are consistent, the parties perform the following steps:

- Derive $[[\tau]] \equiv \langle [\tau], [\theta_\tau], [\delta_\tau] \rangle$, where
  - $[\tau] \leftarrow \sum_{i=1}^{\kappa} [\tau_i]$
  - $[\theta_\tau] \leftarrow \sum_{i=1}^{\kappa} [\theta_{\tau_i}]$
  - $[\delta_\tau] \leftarrow [\delta]$
- Call $\mathcal{F}_{\text{verfy}}([[\tau]])$: If verification fails, we conclude malicious behavior occurred during protocol execution.

To see why the above verification works, we need to examine closely how the shares are derived. The security guarantee of the verification procedure is formalized in the next section.

4.4 The SC$_m$ Protocol

Protocol 5 SC$_m$ $(\langle [[a]], [[b]] \rangle) \rightarrow [[c_0]]$

```
Require:
    (a) $\mathcal{F}_{\text{sc}}$ is secure comparison functionality defined in Equation 1.
    (b) $\kappa$ is a parameter indicating $\kappa$-bit statistical security.
    (c) Index $i$ is bounded by $\kappa$, i.e., $1 \leq i \leq \kappa$.
1: Input randomization and replication:
    (a) $\langle [[x]], [[y]], [t_i] \rangle \leftarrow \mathcal{F}_{\text{input\_rand}}([[a]], [[b]])$
2: Input verification:
    (a) For each $\langle [[x]], [[y]], [t_i] \rangle$, call $\mathcal{F}_{\text{verify}}([[x]])$ and $\mathcal{F}_{\text{verify}}([[y]])$
    (b) The protocol aborts if any verification aborted or failed.
3: Executing $\kappa$ secure comparison in parallel and generate authenticated shares for
   each result:
    (a) $[z_i] \leftarrow \mathcal{F}_{\text{sc}}\left(\langle [x], [y] \rangle_{t_i}\right)$
    (b) $[\delta] \leftarrow \mathcal{F}_{\text{rand}}(p)$ and $\left[\delta^{-1}\right] \leftarrow \mathcal{F}_{\text{invert}}([\delta])$
    (c) $[\theta_{z_i}] \leftarrow \mathcal{F}_{\text{mult}}([z_i], [\delta])$ and $[\theta_{t_i}] \leftarrow \mathcal{F}_{\text{mult}}([t_i], [\delta])$
    (d) $[[z_i]] \leftarrow \langle [z_i], [\theta_{z_i}], [\delta] \rangle$
        $[[t_i]] \leftarrow \langle [t_i], [\theta_{t_i}], [\delta] \rangle$
4: De-randomize the results and derive the authenticated shares of de-randomized
   results:
    (a) $[\tau_i] \leftarrow [t_i] + [z_i] - 2 \times \mathcal{F}_{\text{mult}}([t_i], [z_i])$
    (b) $[\zeta_{\tau_i}] \leftarrow 2 \times \mathcal{F}_{\text{mult}}([\theta_{t_i}], [\theta_{z_i}])$
    (c) $[\theta_{\tau_i}] \leftarrow [\theta_{t_i}] + [\theta_{z_i}] - \mathcal{F}_{\text{mult}}\left([\zeta_{\tau_i}], \left[\delta^{-1}\right]\right)$
    (d) $[[\tau_i]] \leftarrow \langle [\tau_i], [\theta_{\tau_i}], [\delta] \rangle$
5: Verifying the result:
    (a) $[\tau] \leftarrow \sum_{i=1}^{\kappa} [\tau_i]$, $[\theta_\tau] \leftarrow \sum_{i=1}^{\kappa} [\theta_{\tau_i}]$, and $[\delta_\tau] \leftarrow [\delta]$
    (b) Call $\mathcal{F}_{\text{verfy}}([[\tau]])$, and the protocol aborts if any verification returns either
        abort or fail.
6: Derive the final authenticated result:
    (a) $[c_0] \leftarrow \kappa^{-1}[\tau]$ and $[\theta_{c_0}] \leftarrow \kappa^{-1}[\theta_\tau]$
    (b) $[[c_0]] \leftarrow \langle [c_0], [\theta_{c_0}], [\delta_{c_0}] \rangle$ where $[\delta_{c_0}] \leftarrow [\delta]$
```

Based on the proposed transformation techniques, we are ready to construct the SC$_m$ protocol whose key steps are presented in Protocol 5. In what follows, we discuss its steps and the intuition behind the proposed design choices. Detailed security analysis is presented in Section 4.5. The protocol takes authenticated shares from the clients or dealers. In real applications, a and b generally belong to two different clients. The shares are distributed to the servers or parties who perform the secure computations.

- **Step 1**: the original input shares are randomized and replicated κ times. Each time, a shared random bit t_i is generated to produce randomized input shares $[[x]]$ and $[[y]]$. Note that those are authenticated shares, but the shares of t_i are not. In the later steps, the protocol will generate authenticated shares for t_i. It is possible to generate the authenticated shares for t_i at this step.

- **Step 2**: verify the integrity of the shares. The protocol aborts if any verification failed or aborted. This step captures if any malicious parties modified their shares during input randomization process.

- **Step 3**: for each randomized input pairs, call the $\mathcal{F}_{\text{sc}}$ functionality, and the randomized comparison result is secretly shared and stored in $[z_i]$. Next, the authenticated shares for both z_i and t_i are produced using a randomly generated and secretly shared value δ.

- **Step 4**: de-randomize the result to obtain the actual result τ_i for each comparison. Step 4(a) performs a secure xor of z_i and t_i; that is, $[\tau_i] = [z_i \oplus t_i]$. Then based on the authenticated shares $[[z_i]]$ and $[[t_i]]$, the protocol derives the authenticated shares of τ_i. Step 4(b) derives $[\zeta_{\tau_i}]$ which will be used to compute $[\theta_{\tau_i}]$ as follows:

$$\begin{aligned}
[\theta_{\tau_i}] &= [\theta_{t_i}] + [\theta_{z_i}] - [\zeta_{\tau_i}][\delta^{-1}] \\
&= [t_i\delta] + [z_i\delta] - 2[t_i\delta][z_i\delta][\delta^{-1}] \\
&= [t_i\delta + z_i\delta - 2t_iz_i\delta] \\
&= [(t_i + z_i - 2t_iz_i)\,\delta] \\
&= [\tau_i\delta]
\end{aligned}$$

  At the end of this step, $[\delta_{\tau_i}]$ is set to $[\delta]$.

- **Step 5**: verify the comparison result based on the strategy discussed in Section 4.3. The protocol aborts if the verification aborted or failed.

- **Step 6**: derive the final authenticated comparison result. κ^{-1} indicates the multiplicative inverse of κ in $\mathbb{Z}_p$. If the verification passed at Step 5, τ is either 0 or κ and θ_τ is either 0 or $\kappa\delta_\tau$. That is:

$$[[\tau]] = \begin{cases} \langle [0], [0], [\delta] \rangle & \text{if } \tau = 0 \\ \langle [\kappa], [\kappa\delta], [\delta] \rangle & \text{if } \tau = \kappa \end{cases}$$

  Thus, after multiplying κ^{-1} with $[\tau]$ and $[\theta_\tau]$, c_0 is set to 0 or 1, and $[[c_0]]$ is returned as

$$[[c_0]] = \begin{cases} \langle [0], [0], [\delta] \rangle & \text{if } c_0 = 0 \\ \langle [1], [\delta], [\delta] \rangle & \text{if } c_0 = 1 \end{cases}$$

At the end of the protocol, each party sends its shares of $[[c_0]]$ to the client whose reconstructs c_0, θ_{c_0} and δ_{c_0}. The client accepts the result if $\theta_{c_0} = c_0\delta_{c_0}$. If SC_m is adopted as a sub-protocol, then $[[c_0]]$ can be directly used for the subsequent secure computations.

The protocol can be simplified by requiring the client to perform some extra computations. At Step 3, the shares of κ pairs of $[[z_i]]$ and $[[t_i]]$ can be returned to the client who then verifies the authenticated shares and de-randomize the comparison results. The client accepts the result if all verifications passed and all κ de-randomized results are the same. Alternatively, if SC_m is a subroutine, the execution could end at Step 4 where the $[[\tau_i]]$ values would be used for the subsequent computations.

4.5 Security and Complexity Analyses

In this section, we analyze the security of SC_m using the real-ideal paradigm. First, we prove the following claim which is related to the correctness of the output verification step of SC_m.

Claim 1. *If the parties follow the SC_m protocol, $\mathcal{F}_{verfy}([[\tau]])$ will succeed. If any shares are modified, the verification will fail with probability*

$$\min\left(1 - \frac{1}{p}, 1 - \frac{1}{2^\kappa}\right)$$

Proof. Suppose the parties follow the protocol to derive $[[\tau]]$ correctly, then all τ_i values are either 0 or 1, and we have

$$\theta_\tau = \sum_{i=1}^{\kappa} \theta_{\tau_i} = \sum_{i=1}^{\kappa} \tau_i\delta = \kappa\tau_i\delta \tag{2}$$

If $\tau_i = 0$, then $\tau = 0$ and $\theta_\tau = 0$. If $\tau_i = 1$, then $\tau = \kappa$ and $\theta_\tau = \kappa\delta$. In either case, θ_τ is derived correctly. Thus, the verification will go through successfully.

Next we analyze the probability that the verification passes when any shares could be modified by the adversary $\mathcal{A}$. During the verification process, the parties check if $\tau\delta = \theta_\tau$. Since t_i is random chosen, z_i is a random. Although τ_i derived at step 4(a) is no longer random, any modifications to the shares of τ_i will make $\tau_i\delta = \theta_{\tau_i}$ highly unlikely because θ_{τ_i} is derived independently from τ_i. In order words, we do not directly compute $\tau_i\delta$ to derive θ_{τ_i}. Instead, it is computed from θ_{t_i}, θ_{z_i} and δ. As a result, if $\tau \neq 0$ or $\tau \neq \kappa$, but still making the equality $\tau\delta_\tau = \theta_\tau$ valid is difficult for the adversary due to the above reasons. The probability of the equality is valid after any modifications to the shares is bounded by $\frac{1}{p}$.

Also, if z_i and t_i could be correctly predicted, the verification would pass. Nevertheless, since each pair of input was randomized, and t_i was randomly generated, the probability that the adversary can predict both values correctly is bounded by $\frac{1}{2^\kappa}$ because there are κ input pairs and t_i values. Combining these probabilities, the overall probability of passing the verification when τ is neither 0 or κ is $\max\left(\frac{1}{p}, \frac{1}{2^\kappa}\right)$ which leads to the claimed failure probability. $\square$

Claim 2. *Protocol SC_m securely implements $\mathcal{F}_{sc_m}$ with abort in the $(\mathcal{F}_{input\_rand}, \mathcal{F}_{verify}, \mathcal{F}_{sc}, \mathcal{F}_{rand}, \mathcal{F}_{invert}, \mathcal{F}_{verify\_zero}, \mathcal{F}_{mult})$-hybrid model in presence of a malicious adversary controlling at most $n-1$ parties.*

Proof. We prove the security of SC_m under the universally composable security model [12]. The main idea is to build a simulator that interacts with the ideal functionalities. If the protocol is secure, then the environment cannot distinguish a real execution of SC_m from an ideal execution between the simulator and the functionalities. The simulator $\mathcal{S}_{\mathrm{sc}_m}$ is constructed as follows:

- Receive $[[a]]^{\mathcal{A}}$ and $[[b]]^{\mathcal{A}}$ from the adversary $\mathcal{A}$.
- $\mathcal{S}_{\mathrm{sc}_m}$ calls the simulator $\mathcal{S}_{\mathrm{input\_rand}}\left([[a]]^{\mathcal{A}}, [[b]]^{\mathcal{A}}\right)$ of $\mathcal{F}_{\mathrm{input\_rand}}$, and let $\left\langle [[x^*]]^{\mathcal{A}}, [[y^*]]^{\mathcal{A}}, [t_i^*]^{\mathcal{A}} \right\rangle$ be the i-th output from $\mathcal{S}_{\mathrm{input\_rand}}$, where $1 \leq i \leq \kappa$.
- For each $\left\langle [[x^*]]^{\mathcal{A}}, [[y^*]]^{\mathcal{A}}, [t_i^*]^{\mathcal{A}} \right\rangle$, $\mathcal{S}_{\mathrm{sc}_m}$ calls the simulators $\mathcal{S}_{\mathrm{verify}}\left([[x^*]]^{\mathcal{A}}\right)$ and $\mathcal{S}_{\mathrm{verify}}\left([[y^*]]^{\mathcal{A}}\right)$ of $\mathcal{F}_{\mathrm{verify}}$. $\mathcal{S}_{\mathrm{sc}_m}$ outputs `abort` if any $\mathcal{S}_{\mathrm{verify}}$ returned `abort`.
- For each $\left\langle [[x^*]]^{\mathcal{A}}, [[y^*]]^{\mathcal{A}}, [t_i^*]^{\mathcal{A}} \right\rangle$ $\mathcal{S}_{\mathrm{sc}_m}$ calls the simulator $\mathcal{S}_{\mathrm{sc}}\left(\left\langle [[x^*]]^{\mathcal{A}}, [[y^*]]^{\mathcal{A}} \right\rangle_{t_i^*}\right)$ of $\mathcal{F}_{\mathrm{sc}}$, and let $[z_i^*]^{\mathcal{A}}$ be the output of each call to $\mathcal{S}_{\mathrm{sc}}$.
- $\mathcal{S}_{\mathrm{sc}_m}$ calls the simulator $\mathcal{S}_{\mathrm{rand}}(p)$ of $\mathcal{F}_{\mathrm{rand}}$ to produce $[\delta^*]^{\mathcal{A}}$, and calls the simulator $\mathcal{S}_{\mathrm{invert}}\left([\delta^*]^{\mathcal{A}}\right)$ of $\mathcal{F}_{\mathrm{invert}}$ to produce $\left[\delta^{-1*}\right]^{\mathcal{A}}$.
- $\mathcal{S}_{\mathrm{sc}_m}$ calls the simulator $\mathcal{S}_{\mathrm{mult}}\left([z_i^*]^{\mathcal{A}}, [\delta^*]^{\mathcal{A}}\right)$ and $\mathcal{S}_{\mathrm{mult}}\left([t_i^*]^{\mathcal{A}}, [\delta^*]^{\mathcal{A}}\right)$ of $\mathcal{F}_{\mathrm{mult}}$. Let $\left[\theta_{z_i}^*\right]^{\mathcal{A}}$ and $\left[\theta_{t_i}^*\right]^{\mathcal{A}}$ be the output of $\mathcal{S}_{\mathrm{mult}}$.
- $\mathcal{S}_{\mathrm{sc}_m}$ calls $\mathcal{S}_{\mathrm{mult}}\left([t_i^*]^{\mathcal{A}}, [z_i^*]^{\mathcal{A}}\right)$, $\mathcal{S}_{\mathrm{mult}}\left(\left[\theta_{t_i}^*\right]^{\mathcal{A}}, \left[\theta_{z_i}^*\right]^{\mathcal{A}}\right)$, $\mathcal{S}_{\mathrm{mult}}\left(\left[\zeta_{\tau_i}^*\right]^{\mathcal{A}}, \left[\delta^{-1*}\right]^{\mathcal{A}}\right)$. From the outputs, $\mathcal{S}_{\mathrm{sc}_m}$ derives $[[\tau_i^*]]^{\mathcal{A}}$ and $[[\tau^*]]^{\mathcal{A}}$.
- $\mathcal{S}_{\mathrm{sc}_m}$ calls $\mathcal{S}_{\mathrm{verify}}\left([[\tau^*]]^{\mathcal{A}}\right)$. $\mathcal{S}_{\mathrm{sc}_m}$ outputs `abort` if $\mathcal{S}_{\mathrm{verify}}$ returned `abort`.
- $\mathcal{S}_{\mathrm{sc}_m}$ sends $[[\tau^*]]^{\mathcal{A}}$ to $\mathcal{A}$, and outputs whatever $\mathcal{A}$ outputs.

It is straightforward to check that the simulated view produced by $\mathcal{S}_{\mathrm{sc}_m}$ is computationally indistinguishable from the real view. □

5 Conclusion and Future Work

In this paper, we present a compiler that is capable of transforming most passive secure comparison (SC) protocols into the ones against malicious adversaries. The compiler has the following benefits:

- It is general and works for most existing passive secure SC protocols.
- We only need to adjust the value of κ to achieve covert security without changing the protocol implementations.
- It is more efficient than the cut-and-choose technique to achieve covert security since the compiler requires less number of copies than the standard cut-and-choose technique.

As a future work, we will benchmark the performance of our compiler against the current state of the art.

References

[1] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In Salil P. Vadhan, editor, *Theory of Cryptography*, pages 137–156, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[2] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, April 7 2009.

[3] Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using topgear in overdrive: A more efficient zkpok for spdz. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography – SAC 2019*, pages 274–302, Cham, 2020. Springer International Publishing.

[4] Carsten Baum, Ivan Damgård, Tomas Toft, and Rasmus Zakarias. Better preprocessing for secure multiparty computation. In *Applied Cryptography and Network Security*, pages 327–345, Cham, 2016. Springer International Publishing.

[5] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[6] Zuzana Beerliová-Trubíniová and Martin Hirt. Efficient multi-party computation with dispute control. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, pages 305–328, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[7] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In Ran Canetti, editor, *Theory of Cryptography*, pages 213–230, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[8] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online spdz! improving spdz using function dependent preprocessing. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 530–549, Cham, 2019. Springer International Publishing.

[9] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 1–10, New York, NY, USA, 1988. Association for Computing Machinery.

[10] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, Lecture Notes in Computer Science, pages 663–680, Berlin, Heidelberg, 2012. Springer.

[11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS'12)*, pages 309–325, January 2012.

[12] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[13] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 11–19, Chicago, Illinois, USA, January 1988.

[14] David Chaum, Ivan B. Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 87–119, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[15] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 34–64. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science.

[16] Ronald Cramer and Ivan Damgård. On the amortized complexity of zero-knowledge protocols. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, pages 177–191, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[17] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, pages 285–304, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[18] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*, pages 285–304. Springer, 2006.

[19] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Efficient and secure comparison for on-line auctions. In *Information Security and Privacy*, pages 416–430, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[20] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority – or: Breaking the spdz limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security – ESORICS 2013*, pages 1–18, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[21] Ivan Damgård and Jesper Buus Nielsen. Scalable and Unconditionally Secure Multiparty Computation. In Alfred Menezes, editor, *Advances in Cryptology -*

CRYPTO 2007, pages 572–590, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

[22] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 643–662, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[23] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure simd circuits. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, pages 721–741, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

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

[25] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the* 17^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 101–111, September 21 1998.

[26] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 465–482, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[27] Oded Goldreich. *The Foundations of Cryptography*, volume 2, chapter General Cryptographic Protocols. Cambridge University Press, 2004.

[28] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game - a completeness theorem for protocols with honest majority. In *19th ACM Symposium on the Theory of Computing*, pages 218–229, New York, New York, United States, 1987.

[29] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-Efficient Unconditional MPC with Guaranteed Output Delivery. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 85–114, Cham, 2019. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[30] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed-Output Delivery Comes Free in Honest Majority MPC. In *Advances in Cryptology – CRYPTO 2020*. Springer International Publishing, 2020. Series Title: Lecture Notes in Computer Science.

[31] Martin Hirt and Jesper Buus Nielsen. Robust Multiparty Computation with Linear Communication Complexity. In *Advances in Cryptology - CRYPTO 2006*, volume 4117, pages 463–482. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. Series Title: Lecture Notes in Computer Science.

[32] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *The 20th USENIX Security Symposium*, August 2011.

[33] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 145–161, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[34] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1575–1590, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 830–842, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making spdz great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 158–189, Cham, 2018. Springer International Publishing.

[37] Yehuda Lindell and Ariel Nof. A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 259–276, Dallas Texas USA, October 2017. ACM.

[38] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *International Workshop on Public Key Cryptography*, pages 343–360. Springer, 2007.

[39] Pascal Paillier. Public key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - Eurocrypt '99 Proceedings, LNCS 1592*, pages 223–238, Prague, Czech Republic, May 2-6 1999. Springer-Verlag.

[40] Tord Reistad. MULTIPARTY COMPARISON - An Improved Multiparty Protocol for Comparison of Secret-shared Values. In *Proceedings of the International Conference on Security and Cryptography*, pages 325–330, Milan, Italy, 2009. SciTePress - Science and and Technology Publications.

[41] Tord Ingolf Reistad and Tomas Toft. Secret sharing comparison by transformation and rotation. In *International Conference on Information Theoretic Security*, pages 169–180. Springer, 2007.

[42] Microsoft SEAL (release 3.6). `https://github.com/Microsoft/SEAL`, November 2020. Microsoft Research, Redmond, WA.

[43] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[44] Andrew C. Yao. Protocols for secure computation. In *Proceedings of the* 23^{rd} *IEEE Symposium on Foundations of Computer Science*, pages 160–164. IEEE, 1982.

[45] Andrew C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 162–167. IEEE, 1986.