SIM: Secure Interval Membership Testing and Applications to Secure Comparison

Abstract—The offline-online model is a leading paradigm for practical secure multi-party computation (MPC) protocol design that has successfully reduced the overhead for several prevalent privacy-preserving computation functionalities common to diverse application domains. However, the prohibitive overheads associated with secure comparison—one of these vital functionalities—often bottlenecks current and envisioned MPC solutions. Indeed, an efficient secure comparison solution has the potential for significant real-world impact through its broad applications.

This work identifies and presents SIM, a secure protocol for the functionality of interval membership testing. This security functionality, in particular, facilitates secure less-than-zero testing and, in turn, secure comparison. A key technical challenge is to support a fast online protocol for testing in large integer rings while keeping the precomputation tractable. Motivated by the map-reduce paradigm, this work introduces the innovation of (1) computing a sequence of intermediate functionalities on a partition of the input into input blocks and (2) securely aggregating the output from these intermediate outputs. This innovation allows controlling the size of the precomputation through a granularity parameter representing these input blocks’ size—enabling application-specific automated compiler optimizations.

To demonstrate our protocols’ efficiency, we implement and test their performance in a high-demand application: privacy-preserving machine learning. The benchmark results show that switching to our protocols yields significant performance improvement, which indicates that using our protocol in a plug-and-play fashion can improve the performance of various security applications. Our new paradigm of protocol design may be of independent interest because of its potential for extensions to other functionalities of practical interest.

1. Introduction

Privacy-enhancing technologies, such as secure multi-party computation (MPC), are essential for bridging the data utility and privacy chasm. MPC [21], [47] allows mutually distrusting parties to compute over their private data without revealing any non-essential information. From initial conceptual prototypes in calculating prices of Danish sugar beet market [7], evaluating gender pay disparities in Boston [30], detecting tax fraud in Estonia [5], and preventing satellite collisions [23], advances in computer hardware have inspired a recent revolution in MPC research and technologies (see [19]).

As more memory and high-end processors become more available, a leading paradigm for practical MPC protocol design is the offline-online model [4], [9], [12]. [13], [22], [26], [33], [34], [40], which offloads most computationally and cryptographically complex operations to an offline precomputation step. The online phase is a fast protocol that uses the output of the precomputation to perform the intended computation securely. Protocol design in this model proceeds by identifying essential atomic functionalities prevalent across diverse high-impact application domains. For example, atomic functionalities like multiplication, (multi-variate) polynomial evaluation, inner product, and comparison have representative applications in arithmetic circuit evaluation [33], decision-tree evaluation [20], [43], [44], private set intersection [14], [24], [39], and neural network training [1], [27], [35], [40], respectively. Individually, each of these atomic functionalities is sufficient to emulate any computation, especially any other atomic functionality listed above. However, in some cases, custom-built precomputations for individual atomic functionalities have demonstrated the possibility of reducing the security overhead of MPC technologies.

The secure comparison functionality, which compares whether one secret value is greater than another or not, is essential in various privacy-sensitive domains like machine learning [36], [45], [46], data analytics [6], and auctions [2], [7]. Furthermore, it is also essential for domains relying on linear programming and dynamic programming—prospective applications include, for example, optimization and biomedical research, respectively. Practical solutions for secure comparison shall create exciting collaborative opportunities in these application domains by meeting the privacy expectations of the actors. Therefore, it is not surprising that there are many works that present secure comparison as a key building block, such as ABY [16], ABY2.0 [37], ABY3 [35], FALCON [46], AriaNN [40], and SecureNN [45], or focus solely on secure comparison, such as [10], [8], and [34]. This paper presents a versatile framework for designing secure comparison protocols in the offline-online model—allowing for performance optimizations in light of network quality (for example, latency, bandwidth, and throughput) and application-specific considerations.

Ishai et al. [26] offer a simple MPC approach that employs function tables to achieve an efficient online phase by precomputing the function for all possible inputs in the offline phase, such that the online phase is simply a table-lookup. This approach pushes as much computation as possible to the offline phase, resulting in a highly efficient online phase but exponentially high precomputation for typical input sizes. This idea may work for small input sizes, but the size of the function table grows exponentially with the input size, thus making many applications unpractical. Applying the map-reduce framework [15] to [26], this work presents SIM, a general framework to
construct efficient secure computation protocols for secure interval membership testing functionality, in turn, leading to efficient secure protocols for the "less than zero" testing and "greater than" functionalities.

This work explores a construction that leverages problem size reduction to shift many computations to the offline phase, which significantly improves the performance of the online phase compared to other traditional methods while keeping the size of function tables computationally tractable and practical. In our secure comparison protocol, we divide the original input into multiple blocks, perform membership testing on each block, and combine block results into the final answer. These steps significantly reduce the size of function tables, thus making them practical and efficient. To the best of our knowledge, this is the first work that brings the use of precomputed function tables into the realm of practical MPC.

Our construction is in the offline-online model, where a third (offline) party prepares a suitable (input-independent) precomputation for two or more online parties (say) Alice and Bob, respectively. Alice and Bob have private inputs \( x_1 \) and \( x_2 \) for the offline phase of the secure comparison protocol, which are additive secret shares of the input \( x \) (i.e., \( x = x_1 + x_2 \)). At the end of the online protocol between Alice and Bob, they obtain the additive secret shares of the output (bit) of interval testing. Our construction is secure against semi-honest (honest-but-curious) adversaries in a two-online-party setting. Further, we achieve malicious security efficiently by extending our system setting to three or more online parties.

For an \( n \)-bit input, we divide the input into \( k \) blocks of \( \ell \)-bits each such that \( n = k \cdot \ell \). Our online phase of secure membership testing has three communication rounds, and its communication cost is \( k \cdot \ell + 2k + 1 \) bits. The precomputation results in a memory requirement of around \( k(2^\ell + 3^2) \) bits per input. We compare our work with various state-of-the-art comparison protocols, including ABY [16], ABY2.0 [37], ABY3 [35], BLAZE [38], and FALCON [46]. Overall, our work has lower communication complexity and outperforms most works with respect to round complexity.

To demonstrate the performance of our protocols, we implement our protocols on the state-of-the-art FALCON framework [46]. The performance benchmark illustrates that our secure comparison is about \( 2 \times \) faster than FALCON’s secure comparison protocol, with \( 4 \times \) cheaper communication. In a semi-honest setting, by only replacing FALCON’s secure comparison protocol with ours, the neural network training efficiency has improved to around \( 1.3 \times \) for a network with ReLU activation function and around \( 1.25 \times \) for a network with MaxPool activation function. In the malicious setting, we achieve an even larger performance improvement (\( 1.35 \times \) faster and \( 5 \times \) cheaper communication for Network A).

Structure of the paper. In Section 2, we provide a general picture of our protocols and explain how it works with a concrete example. In Section 3, we introduce the background and the system setting of our protocols. Then we formalize the use of function tables in MPC in Section 4.

We formalize and provide the detailed construction of our protocol in Section 5, which is followed by the cost analysis and security analysis (Section 6). In Section 7, we show how to achieve malicious security efficiently. Then in Section 8, we illustrate how our protocol can help with building high-level applications such as privacy-preserving neural network training/inference. Meanwhile, we provide our implementation details and the benchmark results there. In Section 9, we show the potential of our protocols to be generalized in multiple ways. Finally, we cover the related works in Section 10.

2. Solution Overview

We consider computations over the set of integers \( \{-2^{n-1}, \ldots, -1, 0, 1, \ldots, 2^{n-1} - 1\} \) (i.e., a ring \( \mathbb{Z}_{2^n} \) along with integer addition and multiplication modulo \( 2^n \) as the operators). Here, a subset \( I \subseteq \mathbb{Z}_{2^n} \) is an interval if there are \( i, j \in \mathbb{Z}_{2^n} \) such that \( I = [i, i+1, \ldots, j] \). Let indicator function \( 1_I : \mathbb{Z}_{2^n} \to \{0, 1\} \) represent the membership functionality for interval \( I \). For any \( x \in \mathbb{Z}_{2^n} \), we have \( 1_I(x) = 1 \) if \( x \in I \); otherwise, \( 1_I(x) = 0 \). In particular, when \( I = N \) is the set of all “negative elements” for an \( n \)-bit value, where \( N := \{-2^{n-1}, -2^{n-1} - 1, \ldots, -1\} \), the functionality \( 1_N \) tests whether \( x \in \mathbb{Z}_{2^n} \) is “less than zero” or not.

2.1. An Illustrative Example for Secure Less-Than-Zero Computation

Towards illustrating the solution strategy, we first present a simplified example of secure less-than-zero functionality for two parties (Alice and Bob) for \( n = 6 \)-bit inputs. (See Figure 1.) For ease of exposition, assume that the objective of the parties is to determine the answer \( s = 1_N(x) \) in the clear for secret shared input \( x \).

With a uniformly random key \( r \in \mathbb{Z}_{2^n} \) that is hidden from both parties, parties reconstruct the corresponding masked secret \( y = (x + r) \). In the example, we consider \( x = 33 \) and \( r = 1 \), which results in \( y = 34 \) in the clear. Next, we equivalently need to compute bit \( s \) indicating whether the blinded output \( y = (x + r) \) is an element of the interval \( N + r \) (whether \( y \) is in the interval \([33, 63, 0]\)).

Let \( \overline{y} \) represent the \( n = 6 \)-bit (two’s complement) binary representation of the element \( y \) (100010). For illustrative purposes, consider the number of blocks \( k = 3 \) and each block length \( \ell = n/k = 6/3 = 2 \). For brevity, let \( y_1 = \overline{y}_{1, \ell} \) (the most significant \( \ell \) bits of \( \overline{y} \)), \( y_2 = \overline{y}_{1, \ell + 1, 2\ell} \) (the middle \( \ell \)-bits of \( \overline{y} \)), and \( y_3 = \overline{y}_{2\ell + 1, 3\ell} \) (the least significant \( \ell \) bits of \( \overline{y} \)). For this example, \( y_1 = 10, y_2 = 00, y_3 = 10 \).

Level \( 1 \) search. We use the first block to perform a level \( 1 \) search. Consider the interval \( J(y_1) = \{y_1||0^2, \ldots, y_1||1^6\} \subseteq \mathbb{Z}_{2^n} \). intuitively, this interval contains all values that begin with \( y_1 \). Let \( f_1 : \{0, 1\}^\ell \to F_3 \) be the function defined as follows. If the interval \( J(y_1) \subseteq N + r \) then \( y \in N \), we define

1. Note that, with respect to the \( +1 \) operator, the elements of \( \mathbb{Z}_{2^n} \) embed on a size-2\(^n\)-circle \( 0 \to 1 \to \cdots \to 2^{n-1} - 1 \to -2^{n-1} \to \cdots -1 \to 0 \), a one-dimensional torus (informally, a cycle). The mentioned intervals are defined over such a one-dimensional torus. Therefore, if \( i = 1 \) and \( j = -1 \), then the corresponding interval is \( I = \{-1, 0, 1\} \). For \( i = 1 \) and \( j = -1 \), the corresponding interval is \( I = \{1, 2, \ldots, 2^{n-1} - 1, -2^{n-1}, \ldots, -2, -1\} = \mathbb{Z}_{2^n} \setminus \{0\} \).
shall become apparent in the discussion below. Note that this auxiliary information is masked in our starting point of the interval or the end point of the interval or the end point of the interval.

However, if \( x \) is such that \( x \cap \mathbb{N} + r \) is non-empty intersections with both \( N + r \) and \( \mathbb{Z}_2 \setminus (N + r) \), the function \( f_1(y_1) = 2 \) indicating this uncertainty. Intuitively, the function \( f_1 \) is a coarse-grained membership testing functionality (and the granularity parameter \( \ell \) determines the granularity of this search). In this example, we can see that if \( y_1 \) is 11 then \( f_1(y_1) = 1 \), since all values of \( y \) starting with 11 are Negative. If \( y_1 \) is 01 then \( f_1(y_1) = 0 \), otherwise \( f_1(y_1) = 2 \).

Alice and Bob compute the output \( a_1 = f_1(y_1) \) securely using the precomputed function table. However, there is a subtlety. Obtaining the answer \( a_1 \) in the clear reveals additional information about \( x \), rendering the protocol insecure. In particular, if \( x \) is close to either of the end-points of \( N \), then the probability of \( a_1 = 2 \) is high. However, if \( x \) is far from both the end-points of \( N \), then the probability of \( a_1 = 2 \) is low. Therefore, parties obtain \( a_1 \in F_3 \) instead using \( a_1 \) using a random shared \( r_1 \) that is hidden from them. In the concrete example, we omit this subtlety, and instead directly present the answering using \( P, N, \) and \( U \) for Positive, Negative, and Undetermined, respectively.

Observe that there can be at most two values of \( y_1 \) such that \( f_1(y_1) = 2 \). The function \( f_1(y_1) \) also needs to produce an auxiliary information \( b \in \{0, 1\} \), representing whether the uncertainty stems from the inclusion of the starting point of the interval or the end point of the interval. Note that this auxiliary information is masked in our protocol, but we omit the masking in this example for ease of understanding. The utility of this auxiliary information shall become apparent in the discussion below.

**Level 2 search.** Next, parties need to perform a finer-grained search if \( f_1(y_1) = 2 \) using the second block of inputs. Alice and Bob continue to the second level of the search while being oblivious to whether \( f_1(y_1) = 2 \) or not. They test whether the interval \( J(y_1\|y_2) = \{y_1\|y_2|0^i\ldots\},y_1\|y_2|1^i\} \) is entirely inside the interval \( N + r \), entirely outside the interval \( N + r \), or partially intersecting both these sets, indicating uncertainty. We define \( a_2 = f_2(y_2, b) \) to be this function. We emphasize that one needs \( b \) to reconstruct \( y_1 \). Again, parties use precomputed function tables to securely compute \( a_2 \), the masking of \( a_2 \) with a random share \( r_2 \).

In this example, we assume \( b = 1 \). The parties then look up the entry for \( y_2 = 00 \) in the table for the second block with \( b = 1 \). This is testing whether the interval \( \{100000, 100001, \ldots, 100011\} \) is entirely negative, entirely positive, or partially negative and partially positive.

Observe that in this search, irrespective of whether \( b = 0 \) or \( b = 1 \), there can be at most one \( y_2 \) such that \( f_2(y_2) = 2 \). This property is essential to ensure that we do not need to generate any additional auxiliary information for this search; otherwise, the domain of auxiliary information would have increased exponentially in \( k \) (the total number of search levels).

**Level 3 search.** Alice and Bob continue with their final search, oblivious to whether their previous searches are uncertain or not. They test whether the (singleton) interval \( J(y_1\|y_2\|y_3) = \{y_1\|y_2\|y_3\} \) is inside of \( N + r \) or not. These cases are exhaustive because a singleton interval cannot be uncertain. Define the function \( a_3 = f_3(y_3, b) \) to be this function. Parties securely compute the encryption \( \tilde{a}_3 \) (using the secret key \( r_3 \)) of the output \( a_3 \) using precomputed function tables. Note that although we describe these searches in multiple levels, one can perform all \( k \) block searches in parallel as long as the auxiliary information \( b \) is known.

**Reconstruction of answer.** Given the (masked) answers \( \tilde{a}_1, \tilde{a}_2, \tilde{a}_3 \) and using built-in \( r_1, r_2, r_3 \), the aggregation function \( g \) finds the smallest index \( i \in \{1, 2, 3\} \) such that \( a_i \neq 2 \) and \( a_j = 2 \), for all \( j \in \{1, \ldots, i - 1\} \).
Finally, the output of \( g \) is \( s = a_i \). Parties compute this aggregation function securely using another precomputed function table to obtain the output \( s \).

Intuitively, the reconstruction function inherently accounts for the masking of the intermediate outputs and outputs the first value that is not Undetermined.

**Efficiency.** Through our approach, the number of rows in all the function tables is 24. As a comparison, if we directly use one single function table, the number of rows is \( 2^{15} = 64 \). Therefore, our approach significantly reduces the size of precomputed function table. This effect is larger with the increase of \( n \). (e.g. when \( n = 64 \), our approach requires \( 15 \times 2^{6} \) rows, while one single function table requires \( 2^{15} \) rows.)

### 2.2. Protocol Steps of SIM

The above discussed technique extends to any interval \( I \) that satisfies \(|I| \geq 2^{n-\ell} \). We conclude the procedures of secure interval-membership testing below:

1. As \( \mathbb{I}_I(x) = \mathbb{I}_{I+r}(y = x + r) \), for any \( r \in \mathbb{Z}_{2^n} \), where \( I + r \) is the interval \( \{x + r: x \in I\} \), the pre-computation samples an element \( r \leftarrow \mathbb{Z}_{2^n} \) uniformly random, hidden from Alice and Bob. Alice and Bob reconstruct \( y = x + r \) during the online phase.

2. Let \( \tilde{y} \in \{0,1\}^n \) represent the (two’s complement) binary representation of the element \( y \in \mathbb{Z}_{2^n} \). Let \( \tilde{y}_1, \ldots, \tilde{y}_n \in \{0,1\}^\ell \) represent the partition of \( \tilde{y} \) into \( k \) blocks of \( \ell \)-bit strings, i.e., \( \tilde{y} = \tilde{y}_1 \parallel \cdots \parallel \tilde{y}_k \).

3. Let \( a_1, \ldots, a_k \) represent the encryptions of the intermediate outputs \( a_1, \ldots, a_k \), respectively, mentioned above. Alice and Bob interactively aggregate the answer \( s = g(a_1, \ldots, a_k) \) (in a secret-shared manner), where the function \( g: F_{k \times 2^n} \rightarrow \{0,1\} \) depends on the secret keys \( r_1, \ldots, r_k \), and satisfies the identity \( g(a_1, \ldots, a_k) = \mathbb{I}_I(x) \). At the end of the online phase, Alice and Bob, respectively, obtain the additive secret shares \( [s]_1 \) and \( [s]_2 \) of the secret \( s \).

### 3. Preliminaries

#### 3.1. System Model

We consider an asymmetric three-party setting where three parties \( P_1, P_2, P_3 \) perform different tasks for the protocol. \( P_3 \) is only involved in the offline phase and does not participate in the online phase, while \( P_1 \) and \( P_2 \) only participate in the online phase, similar to [41], [42], [48].

2. If \(|I|\) is smaller, then both of its endpoints may fall within one uncertain interval of the level 1 search. Given this possibility, one needs to generate auxiliary information in level 2 search because parties are oblivious to whether this event has already occurred or not. This generation of auxiliary information in every level of the search increases the input domains of the search functions and the aggregation function exponentially in \( k \).

We assume point-to-point authenticated and secure communication channels between the parties. The communication channels are assumed to be bounded-synchronous [3], meaning that the protocol operates in rounds. In each round, parties can perform local computations and send messages. By the end of the round, all parties are guaranteed to receive the messages sent to them in that round. Our protocol also works in the standard offline/online model.

SIM tolerates a semi-honest adversary who can corrupt at most one party. We emphasize that the adversary can corrupt any one of the parties, in particular, the adversary is allowed to corrupt \( P_3 \), who is only involved in the offline phase of the protocol. We extend our protocol against the malicious adversary in Section 7.

#### 3.2. Two-Party Secret Sharing Based MPC

We use a three-party MPC (3PC) setting where two parties participate in the online phase while the third party only produces the offline data and is absent from the online phase. Therefore, our construction closely follows two-party MPC (2PC) protocols for the online phase, and we summarize the relevant 2PC concepts below.

##### 3.2.1. Linear Two-Party Secret Sharing

A linear secret-sharing scheme is a secret-sharing scheme where parties only need to perform local operations on their shares to perform linear operations such as addition on the secret-shared value. In this work, we use a specific linear secret-sharing scheme, known as additive secret sharing. To share a secret \( s \), the first share \([s]_1 \) is generated uniformly at random, and the second share is set such that the sum of the two shares is the secret. To reconstruct the secret \( s \), the two parties add their respective shares together. We can see that additive secret sharing indeed has the linear property. For example, to add two secrets \( s \) and \( t \), parties simply need to locally add their share of \( s \) to their share of \( t \) to obtain their share of \( s + t \).

Formally, the dealer shares a secret \( s \in \mathbb{Z}_{2^n} \) to two parties \( P_1, P_2 \) by generating shares \([s]_1, [s]_2 \in \mathbb{Z}_{2^n} \) such that each party \( P_i \) holds \([s]_i \) and \([s]_1 + [s]_2 = s \). We use \([s] \) to denote the secret sharing of \( s \). Here, for public constants \( a, b \in \mathbb{Z}_{2^n} \), and secret shared values \([s]_i, [u]_i \), the following identity holds: \([a \cdot s + b \cdot u] = a \cdot [s] + b \cdot [u] \).

##### 3.2.2. MPC based on secret sharing

In two-party secret-sharing-based MPC, the function inputs are secret-shared to the two parties \( P_1, P_2 \), where operations are then carried out on the secret shares. When using a linear secret-sharing scheme, the addition of two secret shared values becomes the local addition of the respective shares.

When we need to multiply two values, we can use state-of-the-art techniques such as [4], [16], which require the parties to communicate with each other to perform the multiplication of two secret shared values.

#### 3.3. Offline-Online Model and Secure Precomputed Function (SPF) Tables

For our secure comparison construction, we follow the standard offline-online model of MPC, where the computation is separated into an (input-independent) offline
phase and an online phase. In the offline phase, input-independent correlated secret sharings are generated that are consumed during the online phase.

As the communication and round complexity of the online phase are critical for the performance of an MPC protocol, our goal is to optimize the online phase and try to push more cryptographically expensive operations to the offline phase. In particular, we build upon the secure precomputed function (SPF) table approach introduced by Ishai et al. [26], which offers a very efficient online phase for computing any function.

The intuition behind the SPF protocol is to precompute the function output for all possible inputs to generate function tables in the offline phase, which allows the online phase to be simple function lookups (which is highly efficient). We formalize the original Ishai et al.’s protocol in Section 4.

This approach manages the small input domains well; however, the size of the SPF table increases exponentially with the input size, and is not practical to use for applications using moderate-sized input domains. While our construction uses SPF tables to achieve a highly efficient online phase, as a key novelty, we significantly reduce the function table size.

4. Formalizing and Extending SPF Compiler

To illustrate how the function table is used in MPC, we first formalize the original (input-hiding) SPF protocol [26] as a compiler. We then introduce our (function-hiding) SPF compiler.

4.1. Original (Input-Hiding) SPF Compiler

Consider a 2PC with two parties $P_1$ and $P_2$ holding inputs $x$ and $y$ respectively, and the parties want to jointly compute $f(x, y)$ for some function $f$. Here, function $f : X_1 \times X_2 \rightarrow Y$, where $(X_1, +), (X_2, +),$ and $(Y, +)$ are groups.

The input-hiding SPF compiler [26] takes function $f$ as the input and outputs an MPC protocol that securely realizes the function. In particular, the compiler takes a matrix $\{f(x_1, x_2)\}_{(x_1, x_2) \in X_1 \times X_2}$ as input and outputs (i) a 2-party correlation $Q_f$ (so-called the precomputation/ offline phase), and (ii) a 2-party online protocol $\Pi_f$. Notice that the compiler depends only on the input-output behavior of the function, and is independent of how the functionality $f$ is realized. To emphasize this property, we use $f$ to mean the input-output behavior of $f$.

Figure 2 describes the generation of the precomputation $Q_f$ and the online protocol $\Pi_f$. In the offline phase, the offline party prepares a random element $r$, then constructs function tables of the input $x+r$ for all $x$ in the input domain. Finally, it secret-shares the SPF table and sends the shared table together with $[r]$ to online parties. In the online phase, online parties add the input $[x]$ and $[r]$ locally, then reconstruct $x+r$. Finally, online parties check the SPF table using $x+r$ to get the final output.

4.2. Function-Hiding SPF Compiler

Towards developing our efficient protocol, we formalize a variant of the input-hiding compiler [26]. Compared to the input-hiding compiler, our function-hiding compiler also takes one round and has a lower communication cost. This variant takes as input a function from a family of functions, and outputs a secure protocol that does not hide the input to the function, but instead hides the function within the family of functions. We call it the function-hiding compiler and use this variant to construct our secure comparison protocol.

Consider a function $f : X \rightarrow Y$, where $(Y, +)$ is a group. Our variant takes the matrix $\{f(x)\}_{x \in X}$ as input and outputs (i) a 2-party correlation $P_f$, and (ii) a 2-party online protocol $\Pi_f$.

Figure 3. Our variant’s procedure to generate the offline protocol $P_f$ and the online protocol $\Pi_f$.

<table>
<thead>
<tr>
<th>Description of the correlation $Q_f$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Pick uniformly random offsets $\alpha_1 \in X_1$ and $\alpha_2 \in X_2$.</td>
</tr>
<tr>
<td>2) Generate $F^{(1)} = {f^{(1)}(x_1, x_2)}_{(x_1, x_2) \in X_1 \times X_2}$, where each element $f^{(1)}(x_1, x_2)$ is uniformly and independently chosen from the group $Y$.</td>
</tr>
<tr>
<td>3) Let $F^{(2)} = {f^{(2)}(x_1, x_2)}_{(x_1, x_2) \in X_1 \times X_2}$ be the unique matrix defined by the identity $f(x_1, x_2) = f^{(1)}(x_1 + \alpha_1, x_2 + \alpha_2) + f^{(2)}(x_1 + \alpha_1, x_2 + \alpha_2)$, for every element $(x_1, x_2) \in X_1 \times X_2$.</td>
</tr>
<tr>
<td>4) Send $(\alpha_1, F^{(1)})$ to Party 1 and $(\alpha_2, F^{(2)})$ to Party 2.</td>
</tr>
</tbody>
</table>

Figure 2. The original compiler’s procedure to generate the offline precomputation $Q_f$ and the online protocol $\Phi_f$.

<table>
<thead>
<tr>
<th>Description of the 2-party protocol $\Phi_f$.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Private inputs.</strong> For $i \in {1, 2}$, Party $i$ has private input $x_i \in X_i$.</td>
</tr>
<tr>
<td><strong>Correlated private randomness.</strong> For $i \in {1, 2}$, Party $i$ has correlated private randomness $(\alpha_i, F^{(i)})$ given by the correlation $Q_f$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description of the protocol.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) For $i \in {1, 2}$, party $i$ broadcasts $x'_i = x_i + \alpha_i$. (Round 1)</td>
</tr>
<tr>
<td>2) For $i \in {1, 2}$, party $i$ outputs $y_i = F^{(i)}(x'_1, x'_2)$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description of the 2-party protocol $\Pi_f$.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Public inputs.</strong> There is a public input $x \in X$ known by all parties.</td>
</tr>
<tr>
<td><strong>Correlated private randomness.</strong> For $i \in {1, 2}$, party $i$ has correlated private randomness $(F^{(i)})$ given by the correlation $P_f$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description of the protocol.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) For $i \in {1, 2}$, party $i$ broadcasts $y_i = F^{(i)}(x)$. (Round 1)</td>
</tr>
<tr>
<td>2) For $i \in {1, 2}$, party $i$ outputs $y = y_1 + y_2$.</td>
</tr>
</tbody>
</table>
phase, the offline party prepares the SPF tables for all possible plaintext input \( x \), then secret-shares it to online parties. In the online phases, the online parties simply check the SPF table using the plaintext input, then reconstruct the final output.

5. Secure Comparison Protocol using Precomputed Function Table

As a showcase of SIM, we introduce our protocol of secure comparison in the form of less-than-zero (LTZ) operation. The protocol takes secret shares \([x]\) as the input, and outputs \([s]\) such that \( s = f_{LTZ}(x) \).

5.1. Protocol Overview

The goal of the protocol is to compute the \( f_{LTZ}(\cdot) \) function for a secret-shared \([x] \in \mathbb{Z}_{2^n}\). First, both online parties compute \([x + r]\) and open it, where \( r \) is a random mask generated during the offline phase. Second, they divide the binary representation of \( x + r \) into \( k \) blocks, where each block has \( \ell \) bits such that \( k \cdot \ell = n \). Third, the online parties perform membership testing functions for all the blocks, and get the intermediate outputs, labeled \( a_1, a_2, \ldots, a_k \). The intermediate outputs can either be Positive(P), Negative(N), or Undetermined(U). As the membership testing function has small input domains, the offline party can precompute the function tables for it such that the online phase of the block membership testing is extremely fast. Finally, the parties open the intermediate outputs and use the intermediate outputs as inputs to the recomputation function table to produce the final output.

In the offline phase, the offline party \( P_0 \) generates the precomputation and function tables needed for the protocol, and secret shares them to two online parties \( P_1 \) and \( P_2 \). We have designed multiple sub-functionalities to collaboratively achieve \( f_{LTZ}(\cdot) \), explained in detail below.

5.2. Building Blocks

We start by introducing the building blocks used in our protocol. Then we explain how to use these functions to construct a full protocol.

We use superscripts to denote public parameters of functions, while subscripts denote secret parameters that need to be kept hidden from the online parties. The secret parameters set during the offline phase are \( r \in \mathbb{Z}_{2^n} \), \( r_1, r_2, \ldots, r_k \in F_3 \), and \( \beta \in \{0, 1\} \), \( r \) is used to mask the input \( x \), \( r_i \) is used to mask the \( i \)-th intermediate result \( z_i \), and \( \beta \) is used to mask the selection bit.

The functions \( SLTZ_{r,\beta} \) are the coarse-grained membership testing functions that test if a block of input can directly tell us the overall output. \( SLTZ_{r,\beta} \) takes public parameter \( i \in \{1, 2, \ldots, k\} \), which represents which of the \( k \) blocks this function is for, and is defined as a family of functions over the secret parameters \( r, r_i \), and \( \beta \).

We start from the first block, the block result is positive if all values that depend on this block (has this block input as prefix) are positive. It is negative if all values that depend on this block are negative, and it is undetermined if some values are positive and some are negative. When the block result of the first block is undetermined, we take the first two blocks as the prefix and check the second block. Since revealing this intermediate result could reveal information on the input, we mask them using the secret parameter \( r_i \).

Formally, we define a function \( SLTZ_{r,\beta}(y_1 y_2 \ldots y_r \bar{b}) : \{0, 1\}^r \times \{0, 1\} \rightarrow F_3 \) such that

\[
SLTZ_{r,\beta}^i(y_1 y_2 \ldots y_r \bar{b}) = \begin{cases} 0 + r_i, & \text{if } z_0 \not\in N + r \text{ and } z_1 \not\in N + r \\ 1 + r_i, & \text{if } z_0 \in N + r \text{ and } z_1 \in N + r \\ 2 + r_i, & \text{otherwise.} \end{cases}
\]

where \( \bar{b} = \text{prefix}_{i, r, \beta = 0}(y_1 y_2 \ldots y_r 00 \ldots 0) \) and \( \bar{z} = \text{prefix}_{i, r, \beta = 0}(y_1 y_2 \ldots y_r 11 \ldots 1) \), and \( \text{prefix}_{i, r, \beta, \bar{b}} \in \{0, 1\}^{n\text{-bits}} \).

Additionally, if \( z_0 \in N + r \) and \( z_1 \notin N + r \), then \( \text{prefix}_{i, r, \beta = 0}(y_1 y_2 \ldots y_r \bar{b}) \). We note that the result of Undetermined can only appear in at most two rows of the function table of the first block. The reason is that the block result is Undetermined if and only if the numbers represented by that row contain both positive numbers and negative numbers. As the numbers in the function table are consecutive and in ascending order, we only have two such cases: When a block contains both 0 and -1, and when a block contains both \( 2^{n-1} - 1 \) and \( -2^{n-1} \).

Since there are at most two uncertain rows in the first block, we build two function tables for the second block (and all the following blocks), one table for each uncertain row in the first block. The input \( \bar{b} \) of \( SLTZ^i_{r,\beta}(y_1 y_2 \ldots y_r \bar{b}) \) is used to indicate if the first group of the tables are used or the second. The goal of \( BLTZ_{r,\beta} \) is to determine \( b \) given the first block of the input, such that \( b \) can be available for following \( SLTZ^i_{r,\beta} \) executions. We use \( \bar{b} \) to represent the randomly masked version of \( b \), and \( b \) will be reconstructed during the online phase.

Formally, we define a function \( BLTZ_{r,\beta}(y_1 y_2 \ldots y_r) : \{0, 1\}^r \rightarrow \{0, 1\} \) such that

\[
BLTZ_{r,\beta}(y_1 y_2 \ldots y_r) = \begin{cases} 0 \oplus \beta, & \text{if } z_0 \in N + r, z_1 \not\in N + r \\ 1 \oplus \beta, & \text{otherwise.} \end{cases}
\]
Algorithm 1: Less Than Zero LTZ([x])

```
Input : [x]
Output : [s]
Pre-computation: [r], PBLTZ, SLTZ, PSLTZ, RECOMB

1. \[ y = [x] + [r] \]
2. \[ b = \Pi_{BLTZ, SLTZ, PBLTZ, PSLTZ} [\gamma_{1, \ell}] \]
3. \[ \hat{b} = \Pi_{BLTZ, SLTZ} [\gamma_{1, \ell}] \]
4. \[ b = \Pi_{BLTZ, SLTZ} [\gamma_{1, \ell}] \]
5. \[ s = \Pi_{RECOMB} (\alpha_i, \beta, \gamma_i) \]
```

Where: \[ \gamma_0 = \frac{y_1 y_2 \ldots y_{n-1}}{n \text{-bits}} \] \[ \gamma_1 = \frac{y_1 y_2 \ldots y_n}{n \text{-bits}} \]

The function \[ RECOMB(\alpha_1, \alpha_2, \ldots, \alpha_k) \] function takes the intermediate outputs produced by SLTZ and combines them to produce the final output. Intuitively, \[ RECOMB(\alpha_1, \alpha_2, \ldots, \alpha_k) \] outputs the first intermediate output that is not undetermined.

Formally, function \[ RECOMB(\alpha_1, \alpha_2, \ldots, \alpha_k) : F^{k}_3 \to \{0, 1\} \]

6. Protocol Analysis

6.1. Cost Analysis and Comparison

As our protocol is technically a 3PC protocol, but the online phase works as a pure 2PC protocol, we present comparisons with state-of-the-art 3PC comparison protocols as well as 2PC protocols.

6.1.1. Three-party Computation Protocols. We first present a fully parameterized theoretical performance comparison in Table 1, then present a few selected parameters to provide concrete performance numbers in Table 2. For our work, the offline phase communication cost consists of three parts: the block function tables (SLTZ), the selection bit table (BLTZ), and the recombination table (RECOMB). BLTZ has \( 2^\ell = 2^\frac{\ell}{\ell} \) rows, where each row has a one-bit element. Each SLTZ table has \( 2^\ell = 2^\frac{\ell}{\ell} \) rows with two-bit elements, and we have in total \( 2k - 1 \) of them (the first block has one SLTZ table, and all following blocks have two SLTZ tables each.). Thus the size of all the SLTZ tables is \( (2k - 1) \cdot 2^\frac{\ell}{\ell} \cdot 2 \). RECOMB table has \( 3^k \) rows, and each row contains an \( n \)-bit secret sharing. The overall offline precomputation communication/storage cost is given by the summation of the three parts above. As for the online communication cost, the reconstruction of \( x + r \) in the first round requires the communication of one \( n \)-bit share. The reconstruction of \( k \) SLTZ results requires \( 2k \) bits of communication. Finally, the reconstruction of the selection bit in the second round takes one-bit of communication.

5. For simplicity, we replace \( \ell \) with \( \frac{\ell}{\ell} \) to keep the number of variables to two.
Theorem 1. Our LTZ protocol securely realizes the ideal functionality \( F_{LTZ} \) in the presence of a PPT adversary \( A \) who can corrupt at most one party as semi-honest.

Proof. To prove this, it suffices to prove Lemma 1 and Lemma 2.

Lemma 1. Our LTZ protocol is secure in the presence of a PPT adversary \( A \) who corrupts \( P_3 \) as semi-honest.

Proof. Since \( P_3 \) is only involved in the input-independent offline phase, adversary \( A \) gains no information.

Lemma 2. There exists a PPT Simulator \( S \) that can simulate the adversary \( A \)'s view in LTZ when \( A \) corrupts either \( P_1 \) or \( P_2 \) such that the simulated view is indistinguishable from the view of the real execution.

Without loss of generality, let \( A \) be a probabilistic polynomial time (PPT) adversary who corrupts \( P_3 \) (since \( P_1 \) and \( P_2 \) are symmetric). We construct a PPT simulator \( S \) that simulates the adversary \( A \)'s view.

To simulate the offline phase, the simulator \( S \) generates uniformly random values as entries for \( P_3 \)'s share of the various function tables as well as for \( P_1 \)'s share of \( r \) and sends them to \( A \). Note that \( S \) records \( P_3 \)'s share of \( r \) as \( r' \).

To simulate the online phase, in the first round, \( S \) samples a uniformly random value as \( [y]_2 \) and sends it to \( A \) to simulate opening \( [y = x + r] \). At the same time, \( S \) receives \( [y]_1 \). \( S \) can recover the input \( [x]_1 \) of \( P_1 \) by computing \( [x]_1 = [y]_1 - r' \) where \( r' \) is recorded from the precomputation step. In the second round, \( S \) samples a uniformly random bit as \( [b]_2 \) and sends it to \( A \) to simulate opening \( [b] \). In the third round, \( S \) samples \( k \) uniformly random element from \( F_3 \) as \( \{\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k\} \) and records them and sends them to \( A \) to simulate opening \( [\hat{a}_1] \). At the same time, \( S \) receives \( \{\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k\} \).

Using \( \{\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k\} \) and \( \{\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k\} \), \( S \) computes \( a_1 = [\hat{a}_1]_1 + [\hat{a}_1]_2 + [\hat{a}_2]_1 + [\hat{a}_2]_2 + \ldots + [\hat{a}_k]_1 + [\hat{a}_k]_2 \). \( S \) now looks at the function table \( RECOMB \) it sent to \( A \) during the offline phase and locates the output of \( \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \) in the function table, labeling it as \( [s]_1 \).

To simulate opening \( [s] \), \( S \) first sends the input \( [x]_1 \) of \( P_1 \) to the ideal functionality, and receives the output \( s' \). \( S \) can then calculate \( [s]_2 \) such that \( [s]_2 = s' - [s]_1 \), and send it to \( A \) to simulate opening \( [s] \).

We now show that this simulation is indistinguishable from the real protocol execution. Since the precomputation that \( P_1 \) receives in the real execution are all secret shares of values, they are all independent and uniformly random, which has identical distribution to what \( S \) generates for precomputation. In the first round, since \( r \) is generated uniformly at random in the real execution, \( x + r \) is
also uniformly at random, and the simulated view has identical distribution to the real execution. In the second round, since $\beta$ is generated uniformly at random in the real execution, $\tilde{b}$ is uniformly random, and the simulated view has identical distribution to the real execution. We highlight the nontriviality that since only one value $b$ is opened from the function tables of $BLTZ_{r,\beta}$, the value of $\tilde{b}$ is independent of anything in the transcript prior to opening $b$. In the third round, since $r_i$ is generated uniformly at random in the real execution, each $\tilde{a}_i$ is also uniformly at random, and the simulated view has identical distribution to the real execution. We once again highlight the nontriviality that only one value from each $SLTZ_{r,\beta,r_i}$ is opened, ensuring that each $r_i$ is effectively only used once, making all $\tilde{a}_i$ independent and uniformly random, which is indistinguishable for the real execution and the simulated view.

Since the final output $\sigma$ is a secret share, it is indistinguishable from a value sampled from the uniform distribution. Note that the $S$ is able to simulate the correct output with random shares, which means the joint distribution of the output and the view is identical for the real execution and the simulation.

We note that this proof is not limited to only LTZ, and can be generalized to other functions mentioned in Section 9.1.

7. Malicious Security

We now present how we can extend our protocol to achieve malicious security.

7.1. System Model

Malicious security can be achieved in a two-online-party setting if we apply the same methods provided in [26], however, it requires a decent amount of additional cost because of the usage of message authentication codes (MACs). Therefore, to achieve a highly-efficient malicious secure protocol, we require a different system model, where one offline party and three online parties are needed. We assume there exists an adversary that can corrupt up to one party. In this honest-majority setting, we can directly use replicated secret sharing to achieve secure-with-abort online phase at no additional cost. This model can be extended such that there are $n \geq 3$ servers and $t < \frac{n}{3}$ corrupted servers.

7.2. Offline Phase

The offline phase of our protocol can be converted to maliciously secure using the standard cut-and-choose technique [31], [32]. Essentially, the offline party prepares $\mu$ sets of precomputations, where $\mu$ is the statistical security parameter. As an additional step to the offline phase, the online parties randomly choose $\mu - 1$ sets of precomputation and open them by sending all values and tables from those $\mu - 1$ sets to each other. The online parties can then locally verify that the function tables and values are generated according to protocol specifications. If any of the sets of precomputation fails the check, the online parties abort and potentially issue punishments towards the offline party. The probability that the offline party can generate an inconsistent offline phase and not be detected is $\frac{1}{\mu}$.

7.3. Online Phase

With the offline phase using the cut-and-choose procedure, we assume that we have one set of precomputed values and that all precomputation is performed according to the protocol specifications.

---

**TABLE 2. COMPARISON OF PERFORMANCE FOR VARIOUS THREE-PARTY PROTOCOLS (WITH $n = 32$, $k = 4$, $\ell = 8$, $\kappa = 128$)**

<table>
<thead>
<tr>
<th></th>
<th>This Work</th>
<th>ABY3 [35]</th>
<th>BLAZE (arithmetic circuit) [38]</th>
<th>BLAZE (garbled circuit) [38]</th>
<th>FALCON [46]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Precomputation)</td>
<td>Offline Communication (bits)</td>
<td>6432</td>
<td>768</td>
<td>288</td>
<td>20544</td>
</tr>
<tr>
<td>Online Communication (bits)</td>
<td>41</td>
<td>576</td>
<td>288</td>
<td>4096</td>
<td>896</td>
</tr>
<tr>
<td>Online Round</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

*The offline cost of FALCON is omitted as we could not obtain concrete numbers.

**TABLE 3. COMPARISON OF PERFORMANCE FOR VARIOUS TWO-PARTY PROTOCOLS (ASSUMING INPUTS ARE $n$-BIT RING ELEMENTS, AND ARE DIVIDED INTO $k$ BLOCKS WITH EACH BLOCK BEING $\ell$ BITS IN OUR PROTOCOL. $\kappa$ IS THE SECURITY PARAMETER OF OBLIVIOUS TRANSFER USED IN GARBLE CIRCUIT PROTOCOLS)**

<table>
<thead>
<tr>
<th></th>
<th>This Work</th>
<th>Garbled Circuit [25]</th>
<th>ABY [16]</th>
<th>ABY2.0 [37]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Precomputation)</td>
<td>Offline Communication</td>
<td></td>
<td>$\kappa$</td>
<td>$4\kappa$</td>
</tr>
<tr>
<td>Communication</td>
<td>$n + 2k + 1$</td>
<td>$n\kappa$</td>
<td>$2n\kappa + n$</td>
<td>$2n\kappa$</td>
</tr>
<tr>
<td>Round</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

*The output of ABY [16] and ABY2.0 [37] are not arithmetic sharing. Therefore some additional sharing conversion (costing one or more rounds) is required if their secure comparison results are used in arithmetic circuits.

**TABLE 4. COMPARISON OF PERFORMANCE FOR VARIOUS TWO-PARTY PROTOCOLS (WITH $n = 32$, $k = 4$, $\ell = 8$, $\kappa = 128$)**

<table>
<thead>
<tr>
<th></th>
<th>This Work</th>
<th>Garbled Circuit [25]</th>
<th>ABY [16]</th>
<th>ABY2.0 [37]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Precomputation)</td>
<td>Memory Usage</td>
<td>6432</td>
<td>128</td>
<td>16384</td>
</tr>
<tr>
<td>Communication</td>
<td>41</td>
<td>4096</td>
<td>8224</td>
<td>8192</td>
</tr>
<tr>
<td>Round</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Since our protocol is in the honest majority setting, it can easily be transformed to tolerate a malicious adversary that controls at most one party. For example, we can use the same replicated secret sharing scheme used in FALCON [46], which will provide security with abort against malicious adversaries. The parties can check if the replicated shares sent by the other two parties are consistent or not. If not, the parties abort. Such a transformation does not affect the cost of our protocol beyond the natural increase in cost due to the maliciously secure framework. For example, replicated secret sharing will require two shares to be sent instead of one during reconstruction but does not otherwise affect our protocol’s performance.

We implement our protocols using FALCON. We refer the readers to Section 8.3 for detailed benchmark and performance analysis.

7.4. Security Analysis

In this section, we informally argue about the security against the malicious adversary.

**Theorem 2.** Assuming the existence of a malicious-secure replicated secret sharing scheme, our LTZ protocol securely realizes the ideal functionality $F_{LTZ}$ in the presence of a PPT adversary $A$ who can corrupt at most one party as malicious under the secure-with-abort setting.

Firstly, the offline-phase is secure due to the cut-and-choose technique. Regrading the online-phase, following the same idea mentioned in Ishai et al. [26], we need to ensure that a malicious party does not choose to report a secret share from a wrong entry in the function table. By using the maliciously-secure replicated secret sharing scheme, the honest parties can detect malicious behavior by reporting inconsistencies between the secret shares of the malicious party and the honest parties. At which point the honest parties can abort the protocol. Then a similar proof to the semi-honest security proof (Theorem 1) applies. Together, our protocol achieves malicious security under the secure-with-abort setting.

8. Application: Privacy-preserving Neural Network Training/Inference

To illustrate the performance of our protocols, we develop prototypes for our secure comparison protocols. Additionally, to demonstrate the effect of our protocol in a real-world application, we choose to implement our protocol for the privacy-preserving neural network training/inference application.

The state-of-the-art works in this area are FALCON [46] and SecureNN [45]. We observe that the main bottleneck is the evaluation of the ReLU activation function, where secure comparison is a core building block. In addition to ReLU, the Division and Maxpool functions also have high costs, with secure comparison as the main reason. Therefore, we think secure comparison is one of the main bottlenecks of privacy-preserving neural network, meaning that the use of our protocol should provide a significant performance improvement. We choose to implement our protocol in C++ and embed it into the FALCON [46] framework by replacing the derivative of ReLU (FALCON’s secure comparison implementation) with our protocol.

8.1. Implementation details

As our protocol has two online parties whereas FALCON has three online parties, we need to properly embed our protocol into the FALCON framework. To begin with, FALCON uses replicated secret sharing. If we only consider the first two parties $P_1$ and $P_2$, they actually hold a two-party additive secret sharing of the input. Therefore, we can run our secure comparison on $P_1$ and $P_2$ without any other changes. However, we still need to make sure the final output $s$ is a three-party replicated share (e.g. $s = s_1 \oplus s_2 \oplus s_3$, and each party $P_i$ holds $s_i$ and $s_{i+1}$). To achieve this, we allow the offline party $P_3$ to generate $s_3$ and $s_1$ in advance and take them as the final shares of $P_3$. In the offline phase, $P_3$ sends $s_1$ to $P_1$ and $s_3$ to $P_2$. In the final recombination table, the shares of $s_2$ are stored instead of the shares of the final output in our original protocol. Therefore, $P_1$ and $P_2$ need an additional round to reconstruct $s_2$ using the shares.

We can think of the LTZ as a black box, FALCON provides 3-party replicated secret sharings as the input to LTZ. Inside the box, LTZ operates with 2-party additive secret sharing, and the output of LTZ is a 3-party replicated secret sharing again. The security of this construction is straightforward in a semi-honest setting.

8.2. Evaluation Results

To compare our protocol against the secure comparison used by FALCON, we choose to evaluate the performances on Network A and Network C from FALCON using the same MNIST [17] dataset that FALCON uses. Network A is a 3 layer fully connected network with the ReLU activation function after each layer. Network C is a 4 layer network with 2 convolutional and 2 fully-connected layers. This network also uses both Max Pooling and ReLU. MNIST [17] is a collection of images of handwritten digits, and is one of the standard dataset used in machine learning literature. For more detail on the dataset, we refer the readers to FALCON [46] and the MNIST [17] dataset.

To begin with, we run the micro-benchmark that only considers the performance of the secure comparison. The micro-benchmark is launched in AWS clusters using three t3.2xlarge instances (8 cores and 32GB RAM). The three instances are located in different regions and have an average round trip time of around 130ms. We call this testing environment the “distributed setting” for the rest of the paper. The experiments show that our LTZ takes 0.790 seconds and 0.163MB of data transmission per party to do 128 secure comparisons. Meanwhile, FALCON’s secure comparison takes 1.353 seconds and 0.606MB bandwidth. So we can expect a 1.7× efficiency improvement in running time and 4× improvements in communication here.

Then we run some neural network tests in the distributed setting. We tested two versions of the codebase: the first one is the original FALCON codebase, and the other is the FALCON codebase with only the LTZ function replaced by our protocol. Therefore all the performance
The cost is acceptable for neural network use cases. The benchmark result is available in Table 5. In general, the neural network training time with our LTZ is around 30% more efficient. For the inference, the running time is improved at around 1.4×. The communication of our protocol is also significantly cheaper than FALCON.

We also provide the execution time and communication of the offline phase in Table 6. The result shows that the cost is acceptable for neural network use cases.

### Table 5. Online Phase Benchmark: Privacy-Preserving Neural Network Training/inference with our LTZ vs FALCON’s LTZ. The experiments are run using AWS t3.2xlarge instances with 130ms ping. Communication is measured by the total MB sent per party. Parameter setting: $n = 32$, $k = 4$, $\ell = 8$.

<table>
<thead>
<tr>
<th>Network</th>
<th>Mode</th>
<th>Online time(s)</th>
<th>Communication (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This work</td>
<td>FALCON</td>
<td>This work</td>
</tr>
<tr>
<td>Network A</td>
<td>training</td>
<td>6.21s</td>
<td>10.14s</td>
</tr>
<tr>
<td></td>
<td>inference</td>
<td>3.16s</td>
<td>1.83s</td>
</tr>
<tr>
<td>Network C</td>
<td>training</td>
<td>22.45s</td>
<td>38.54s</td>
</tr>
<tr>
<td></td>
<td>inference</td>
<td>11.45s</td>
<td>22.39s</td>
</tr>
</tbody>
</table>

**difference** is caused by the replacement of the LTZ function. For the neural network training, we run 15 forward-backward pass iterations just to show the performance difference. The benchmark result is available in Table 5. In general, the neural network training time with our LTZ is around 30% more efficient. For the inference, the running time is improved at around 1.4×. The communication of our protocol is also significantly cheaper than FALCON.

We also provide the execution time and communication of the offline phase in Table 6. The result shows that the cost is acceptable for neural network use cases.

**Table 6. Offline Phase Benchmark for $n = 32$, $k = 4$, $\ell = 8$. (Assuming inputs are $n$-bit ring elements, and are divided into $k$ blocks with each block being $\ell$ bits in our protocol.)**

<table>
<thead>
<tr>
<th>Network</th>
<th>Mode</th>
<th># of Comparisons</th>
<th>Execution Time</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network A</td>
<td>training</td>
<td>606720</td>
<td>780s</td>
<td>488 MB</td>
</tr>
<tr>
<td></td>
<td>inference</td>
<td>34048</td>
<td>44s</td>
<td>28 MB</td>
</tr>
<tr>
<td>Network C</td>
<td>training</td>
<td>19968000</td>
<td>7.16 hr</td>
<td>16.1 GB</td>
</tr>
<tr>
<td></td>
<td>inference</td>
<td>1324800</td>
<td>1688s</td>
<td>1.1 GB</td>
</tr>
</tbody>
</table>

*The benchmark is executed in AWS clusters with t3.2xlarge instances.*

**Tuning Parameters for Better Performance.**

![Offline phase benchmark of 128 secure comparisons](image)

Figure 4. Offline phase benchmark with different parameter settings ($n = 32$, $k$ is the number of blocks). The experiments are run using AWS t3.2xlarge instances with 130ms ping.

Our protocol is highly flexible since the parameters such as $k$ and $\ell$ are all flexible to change, and different parameter combinations lead to different offline/online performances. Therefore, the users of our protocol can pick the proper parameters to fit into their application. As an illustrative example, we test the offline/online performance of different $k, \ell$ combinations using the same testing environment. Notice that when we fix $k$, the best offline performance can be achieved if the size of all the blocks are the same (or closer to each other). The reason is that the size of the recombination table is fixed with $k$, and the offline phase performance is bottlenecked by the largest block table. The size of the largest block table is most optimized if we set the sizes of all blocks to be the same. We follow this setting in our experiments and only include parameter $k$ in the benchmark figure.

Figure 4 shows the offline phase performance with different parameter settings. When $k$ is very small, the size of each block function table is large, thus the offline phase is costly. Similarly, when $k$ is very large, the size of the recombination table is large, leading to an expensive offline phase. Therefore, if the users pursue a more efficient offline phase, an intermediate $k$ is a good option. Figure 5 illustrates the online phase performance. It can be seen that the online performance increases linearly with $k$. The reason is that the communication complexity and computation complexity of our protocol are both linear with $k$. Therefore, the users can pick the smallest $k$ that they can afford (in terms of the offline phase) to get the best online phase performance. As for the overall performance including both the online and offline phases, the pattern is almost the same as the offline phase since the offline phase is the main bottleneck taking around 90% to 95% of the overall cost.

### 8.3. Implementation and Evaluation of the Malicious Secure Version of Our Protocol

We implement the maliciously secure version of our protocol using FALCON [46] in a three-online-party setting. As we only need share reconstructions in the online
phase, we directly use the replicated secret sharing and the corresponding malicious-secure share reconstruction functions provided by FALCON. Accordingly, the offline party generates function tables with replicated secret sharing as the function table output.

We test the performance of the protocols in AWS, where three servers are instantiated as c5.9xlarge (36 cores and 72GB RAM) instances in the same region. For illustrative purposes, we test the performance of both training and inference with Network A. The benchmark result is shown in Table 7. Compared with the semi-honest case, the main difference is the communication cost, which increases significantly in both our protocols and FALCONs protocols. However, we see that our protocol achieves a larger performance gain than the semi-honest case. The communication cost of our protocol is around $5 \times$ cheaper than FALCON in a malicious setting, while in the semi-honest case it is only around $2 \times$. The reason is that the malicious building block significantly increases the cost of secure comparison, and because our secure comparison protocol requires less reconstruction, the performance gain becomes inherently larger.

9. Generalization

9.1. General Functions

The idea of problem size reduction could be generalized to solve more problems beyond less-than-zero. Since we are using function tables for blocks of input, our protocol can be generalized to a class of functions, which we define as block determinable functions.

We define block determinable functions as functions where the overall output of the function is determined by first examining the inputs block by block. More formally, it satisfies the following identity: $f(x_1||x_2||\cdots||x_k) = h(g_1(x_1), g_2(x_2), \ldots, g_k(x_k))$.

Furthermore, to achieve better performance, the function should have an additional property where either the functions $g_i$s have small ranges (preferably one to two bits), or the function $h$ can be performed efficiently through MPC, e.g. secure addition. In addition to Less Than Zero, Hamming distance and parity are also block determinable functions with significant potential for impact through their privacy-preserving applications.

Note that the above block determinable functions follow a strategy similar to the map-reduce paradigm, where functions $g_i$s resemble mapping and function $h$ resembles reduction operation.

We elaborate on two example functions that can be solved by our protocols, secure hamming distance evaluation and secure equal to zero evaluation.

9.1.1. Secure Hamming Distance. Consider a setting when online parties $P_1$ and $P_2$ hold two secret shared bit strings of the same length $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$. These two strings are represented as ring elements in $\mathbb{Z}_{2^n}$. The hamming distance is defined as $h(X, Y) = \sum_{i=1}^{n} x_i \oplus y_i$.

If we build up one precomputed function table to solve this problem, the size of function table is $2^n \times 2^n$ since the function $h(X, Y)$ has two inputs of size $2^n$. This is too large to be used in practice when $n$ is large.

This efficiency problem can be solved by applying the problem size reduction here. Similarly, in the online phase, we can reconstruct randomly masked $X$ and $Y$ and divide them into $k$ pieces where each piece is $\ell$ bit, such that $n = k \times \ell$. Then we build function tables for each piece so that each function table returns the hamming distance of the corresponding piece. The size of each table is $2^\ell$ and there are in total $k$ function tables.

For hamming distance, there is no need to build up a recombination table to combine block results, since the recombination steps can be done by easily summing up all the block results.

If we set $k = \ell = \sqrt{n}$, then the problem size reduction reduces the offline cost from $O(2^n)$ to $O(\sqrt{n} \times 2\sqrt{n})$. Assuming $n = 64$, this is a $2^{53} \times$ improvement.

9.1.2. Secure Equal to Zero. Equal to Zero can be seen as a special case of the Less Than Zero function. Instead of having 3 possible intermediate outputs (Positive, Negative, Unknown), EQZ will have two possible intermediate outputs (True and False). If a block directly determines that the value is not equal to zero, the intermediate output will be False. A value is overall equal to zero if all blocks return True. While we may still need a recombination table, since the intermediate output space is reduced, our recombination table will only have $2^k$ instead of $3^k$ rows.

9.2. Going Beyond Two Online Parties

In this section, we describe how to turn our protocol from a 3PC protocol to include more online parties. In the original protocol, both online parties execute the same computations, the only difference is that the contents of their function tables are different. Therefore, we can leverage it to extend our protocol to accommodate more online parties. If we consider $n$-online parties, the offline party will generate function tables for each online party, the only difference is that the output of function tables will be $(n, n)$ additive secret sharing instead of 2-party secret sharing. The online parties will follow the same online phase, and the only difference is that they will use $(n, n)$ secret sharing scheme to reconstruct the secrets. In this case, the offline party becomes a single point of failure, as the privacy is broken if the adversary knows the randomness generated by the offline party. Therefore, we can only use this idea when there is a trustworthy third party who is willing to take care of the offline phase.

As an alternative, in some applications, we can also let the clients who provide private inputs generate the precomputation. If the clients don’t generate correct precomputation or leak it to the adversary, the privacy/correctness of their own data will be broken, thus they have no motivation to become a “corrupted offline party”. In this setting, we don’t need an offline party anymore and the precomputation will be taken as part of the inputs of the MPC protocol. For instance, this idea could be used when the online parties hold a neural network model privately in the form of secret sharing, and clients want their private input to be evaluated on the private neural network.

7. $(n, \ell)$ ring-based secret sharing could also be used.
10. Related Works

In [18], Escudero et al. present an efficient secure comparison protocol through the use of novel precomputation values called edabits. The communication complexity of the protocol is $O(n)$, and in our case, we only need to reconstruct approximately two $n$-bit ring elements. Additionally, their round complexity is logarithmic while our protocol is constant round. An alternate way of using edabits is briefly mentioned, in which edabits are used to convert an arithmetic secret sharing to a garbled circuit setting. While this reduces the round complexity, it greatly increases the communication complexity due to the use of garbled circuits.

In [34], Makri et al. uses an adaptation of the BIT-LT protocol by Damgård et al. [11] and edabits [18] to compare a secret-shared value with a public value (Less Than Constant protocol). Such a comparison protocol requires $\log n + 2$ rounds when our protocol requires only constant rounds. Additionally, Makri et al. presented an optimized protocol for LTZ (presented as ReLU). However, the optimized LTZ protocol still requires $\log n$ rounds.

Ryffel et al. [40] extend [9] to perform secure comparison with similar performance using function secret sharing. Compared with them, our protocol provides both the semi-honest version and the malicious secure version while AriaNN only realizes solutions for the semi-honest setting. Besides, their construction achieves computational security while our solution is perfectly secure. Additionally, their protocol has a non-zero error rate. Although in their setting, the error rate is low enough to not significantly affect their application, the error rate may be larger and more significant for other applications and settings. In comparison, our protocol always outputs a correct answer.

There are also plenty of research focusing on privacy-preserving machine learning, thus the secure comparison protocol is one of the key building blocks in these works. For instance, ABY3 [35] uses bit extraction to get the most significant bit of an element, which costs $O(\log(n))$ rounds and $O(n)$ reconstructions. Many works afterward such as SWIFT [28] follow this method to do secure comparison. Additionally, SWIFT reports an amortized online communication cost of $\Omega(n)-6$ bits, which is also higher than our protocol.

ABY [16] and ABY2.0 [37] make use of a parallel prefix adder (PPA) to achieve the extraction of the most significant bit. The circuit itself has $O(\log n)$ depth, thus constant round complexity can only be achieved through the garble circuit method. As a result, ABY2.0 designs efficient protocols to transfer secret sharing from arithmetic form and garble circuit form efficiently. However, this secure comparison requires more communication than our protocol due to the use of the garble circuits. Besides, our protocol works only in the arithmetic world, therefore it can be easily generalized to include more online servers. This generalization is not straightforward for ABY2.0.

Damgård et al. [13] present some efficient building blocks for privacy-preserving machine learning. Their secure comparison protocol follows a similar idea as edabits [18], where a random share and its bit-wise shares are used to accelerate the computation. Their protocol has $O(\log(n))$ round complexity as a bit-wise less than operation is required. Compared to their work, our protocol only takes three rounds for arbitrary size inputs.

Most relevant to our work, SecureNN achieves secure comparison through the use of share conversion between the original even ring to an odd ring in order to extract the most significant bit (MSB) of a value, which directly determines its sign. This construction requires opening $O(n)$ secret sharings in the online phase, thus it is outperformed by our protocol in communication cost. FALCON performs secure comparison (called Derivative of ReLU in their paper) through the use of wrap functions and local computations. The wrap function is essentially a function that computes the carry bit when shares are added together. For a $n$ bit input, both SecureNN and FALCON require $O(\log(n))$ rounds and $O(n)$ bits of communication.

Tetrad [29] follows the direction of ABY2.0, where they construct a mixed-protocol framework and use a garble circuit to extract the most significant bit. Compared with their solution, our protocol is built in pure arithmetic circuits, thus no share transfer is needed. Meanwhile, our protocol can be generalized to $n$ parties much more easily.

11. Conclusion

In this work, we have proposed SIM, a secure interval testing protocol that leverages function tables to achieve both an efficient online phase and a practical offline phase. As the first step, we introduced the abstract functionality of testing membership in an interval. Similar to the map-reduce methodology, we have presented a solution approach for partitioning input into input blocks, computing a sequence of intermediate functionalities on those input blocks, and securely aggregating the output from these intermediate outputs. We have illustrated that our protocol significantly improves the performance of high-level applications such as privacy-preserving machine learning.

As a step towards generalization, we have defined a notion of block determinable functions and also proposed an approach for going beyond two online parties. We believe the proposed techniques have the potential to be applied to other MPC problems as well as generalized to further settings and may be of interest independent of the less-than-zero function.

<table>
<thead>
<tr>
<th>Network Mode</th>
<th>Network A</th>
<th>Online time(s)</th>
<th>Communication (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td></td>
<td>This work</td>
<td>FALCON</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.83s</td>
<td>5.18s</td>
</tr>
<tr>
<td>Inference</td>
<td></td>
<td>0.218s</td>
<td>0.298s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>58MB</td>
<td>216MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.6MB</td>
<td>10.5MB</td>
</tr>
</tbody>
</table>

TABLE 7: MALICIOUS-SECURITY BENCHMARK: PRIVACY-PRESERVING NEURAL NETWORK TRAINING/INFERENCE WITH OUR LTZ VS FALCON’S LTZ. THE EXPERIMENTS ARE RUN USING AWS C5.9XLarge INSTANCES LOCATED IN THE SAME REGION. COMMUNICATION IS MEASURED BY THE TOTAL MB SENT PER PARTY. PARAMETER SETTING: $n = 32$, $k = 4$, $\ell = 8$. |
References


A. Security Proof for General Function

We now provide the security proof for a general function $f$ with the form $f(x) = h(g_1(x_1), g_2(x_2), \ldots, g_k(x_k))$.

**Theorem 3.** Our protocol securely realizes the ideal functionality $F_f$ in the presence of a PPT adversary $A$ who can corrupt at most one party as semi-honest.

**Proof.** To prove this, it suffices to prove Lemma 4 and Lemma 3.

**Lemma 3.** Our protocol is secure in the presence of a PPT adversary $A$ who corrupts $P_3$ as semi-honest.

**Proof.** Since $P_3$ is only involved in the input-independent offline phase, adversary $A$ gains no information.

**Lemma 4.** There exists a PPT Simulator $S$ that can simulate the adversary $A$’s view in our protocol when $A$ corrupts either $P_1$ or $P_2$ such that the simulated view is indistinguishable from the view of the real execution.

Without loss of generality, let $A$ be a probabilistic polynomial time (PPT) adversary who corrupts $P_3$ (since $P_3$ and $P_2$ are symmetric). We construct a PPT simulator $S$ that simulates the adversary $A$’s view.

To simulate the offline phase, the simulator $S$ generates uniformly random values as entries for $P_1$’s share of the various function tables as well as for $P_1$’s share of $r$ and sends them to $A$. Note that $S$ records $P_1$’s share of $r$ as $r^\prime$.

To simulate the online phase, $S$ samples a uniformly random values as $[y_i]_2$ and sends it to $A$ to simulate opening $[y = x + r]$. At the same time, $S$ receives $[y]$ and can recover the input $[x]$ of $P_1$ by computing $[x]_1 = [y] - r^\prime$ where $r^\prime$ is recorded from the precomputation step. Next, $S$ samples $k$ uniformly random elements as $[a_{12}]_2, [a_{22}]_2, \ldots, [a_{k2}]_2$ and records and sends them to $A$ to simulate opening $[a]_2$. At the same time, $S$ receives $[a_{11}]_2, [a_{21}]_2, \ldots, [a_{k1}]_2$.

Using $[a_{11}]_2, [a_{12}]_2, \ldots, [a_{k1}]_2$ and $[a_{12}]_2, [a_{22}]_2, \ldots, [a_{k2}]_2$, $S$ computes $a_i = [a_{i1}]_2 + [a_{i2}]_2 + \ldots + [a_{ik}]_2$. $S$ now looks at the function table for $h(\cdot)$ it sent to $A$ during the offline phase, and locates the output of $a_1, a_2, \ldots, a_k$ in the function table, labels it $[a]_2$. 

---


[38] Qizhi Zhang, Lichun Li, Shan Yin, and Juanjuan Sun. Mpc protocol for g-module and its application in secure compare and relu, 2021.

---

**A. Security Proof for General Function**

We now provide the security proof for a general function $f$ with the form $f(x) = h(g_1(x_1), g_2(x_2), \ldots, g_k(x_k))$.

**Theorem 3.** Our protocol securely realizes the ideal functionality $F_f$ in the presence of a PPT adversary $A$ who can corrupt at most one party as semi-honest.

**Proof.** To prove this, it suffices to prove Lemma 4 and Lemma 3.

**Lemma 3.** Our protocol is secure in the presence of a PPT adversary $A$ who corrupts $P_3$ as semi-honest.

**Proof.** Since $P_3$ is only involved in the input-independent offline phase, adversary $A$ gains no information.

**Lemma 4.** There exists a PPT Simulator $S$ that can simulate the adversary $A$’s view in our protocol when $A$ corrupts either $P_1$ or $P_2$ such that the simulated view is indistinguishable from the view of the real execution.

Without loss of generality, let $A$ be a probabilistic polynomial time (PPT) adversary who corrupts $P_3$ (since $P_3$ and $P_2$ are symmetric). We construct a PPT simulator $S$ that simulates the adversary $A$’s view.

To simulate the offline phase, the simulator $S$ generates uniformly random values as entries for $P_1$’s share of the various function tables as well as for $P_1$’s share of $r$ and sends them to $A$. Note that $S$ records $P_1$’s share of $r$ as $r^\prime$.

To simulate the online phase, $S$ samples a uniformly random values as $[y_i]_2$ and sends it to $A$ to simulate opening $[y = x + r^\prime]$. At the same time, $S$ receives $[y]$.

---

**Proof.** To prove this, it suffices to prove Lemma 4 and Lemma 3.

**Lemma 3.** Our protocol is secure in the presence of a PPT adversary $A$ who corrupts $P_3$ as semi-honest.

**Proof.** Since $P_3$ is only involved in the input-independent offline phase, adversary $A$ gains no information.

**Lemma 4.** There exists a PPT Simulator $S$ that can simulate the adversary $A$’s view in our protocol when $A$ corrupts either $P_1$ or $P_2$ such that the simulated view is indistinguishable from the view of the real execution.

Without loss of generality, let $A$ be a probabilistic polynomial time (PPT) adversary who corrupts $P_3$ (since $P_3$ and $P_2$ are symmetric). We construct a PPT simulator $S$ that simulates the adversary $A$’s view.

To simulate the offline phase, the simulator $S$ generates uniformly random values as entries for $P_1$’s share of the various function tables as well as for $P_1$’s share of $r$ and sends them to $A$. Note that $S$ records $P_1$’s share of $r$ as $r^\prime$.

To simulate the online phase, $S$ samples a uniformly random values as $[y_i]_2$ and sends it to $A$ to simulate opening $[y = x + r^\prime]$. At the same time, $S$ receives $[y]$. $S$ can recover the input $[x]$ of $P_1$ by computing $[x]_1 = [y] - r^\prime$ where $r^\prime$ is recorded from the precomputation step. Next, $S$ samples $k$ uniformly random elements as $[a_{12}]_2, [a_{22}]_2, \ldots, [a_{k2}]_2$ and records and sends them to $A$ to simulate opening $[a]_2$. At the same time, $S$ receives $[a_{11}]_2, [a_{21}]_2, \ldots, [a_{k1}]_2$.

Using $[a_{11}]_2, [a_{12}]_2, \ldots, [a_{k1}]_2$ and $[a_{12}]_2, [a_{22}]_2, \ldots, [a_{k2}]_2$, $S$ computes $a_i = [a_{i1}]_2 + [a_{i2}]_2 + \ldots + [a_{ik}]_2$. $S$ now looks at the function table for $h(\cdot)$ it sent to $A$ during the offline phase, and locates the output of $a_1, a_2, \ldots, a_k$ in the function table, labels it $[a]_2$. 

---
To simulate opening of \([s]\), \(S\) first sends the input \([x]_1\) of \(P_1\) to the ideal functionality, and receive the output \(s^*\). \(S\) can then calculate \([s]_2\) such that \([s]_2 = s^* - [s]_1\), and send it to \(A\) to simulate opening \(s\).

We now show that this simulation is indistinguishable from the real protocol execution.

Since the precomputation that \(P_1\) receives in the real execution are all secret shares of values, they are all independent and uniformly random, which has identical distribution as what \(S\) generates for precomputation. Since \(r\) is generated uniformly at random in the real execution, \(x + r\) is also uniformly at random, the simulated view has identical distribution to the real execution. Since \(r_i\)'s are generated uniformly at random in the real execution, each \(\tilde{a}_i\) is also uniformly at random, the simulated view has identical distribution to the real execution. We highlight the nontriviality that only one value from each \(g(\cdot)\) is opened, ensuring that each \(r_i\) is effectively only used once, making all \(\tilde{a}_i\) independent and uniformly random, which is indistinguishable between the real execution and the simulated view.

Since the final output \([s]\) is a secret share, it is indistinguishable from a value sampled from the uniform distribution. Note that the \(S\) is able to simulate the correct output with random shares, which means the joint distribution of the output and the view is identical for the real execution and the simulation.