Multi-Party Replicated Secret Sharing over a Ring with Applications to Privacy-Preserving Machine Learning

Alessandro Baccarini Marina Blanton Chen Yuan

Department of Computer Science and Engineering
University at Buffalo (SUNY), Buffalo, USA
{anbaccar,mblanton,chyuan}@buffalo.edu

Abstract

Secure multi-party computation has seen significant performance advances and increasing use in recent years. Techniques based on secret sharing offer attractive performance and are a popular choice for privacy-preserving machine learning applications. Traditional techniques operate over a field, while designing equivalent techniques for a ring can boost performance. In this work we develop a suit of multi-party techniques for a ring in the honest majority setting starting from elementary operations to more complex with the goal of supporting general-purpose computation. We demonstrate through empirical evaluation that our techniques can be several times faster than their field-based equivalents and up to two orders of magnitudes faster for certain operations such as matrix multiplication. We also evaluate our techniques on machine learning applications and show that the resulting performance is on par with that of most recent custom protocols for these applications.

1 Introduction

Secure multi-party computation has seen notable performance improvements in recent years that make privacy-preserving computation of increasingly complex functionalities on increasingly large data sets more practical than ever before. Recent significant interest in privacy-preserving machine learning (PPML) has brought to light secret sharing techniques which were often previously overlooked in the literature. Secret sharing offers superior performance for arithmetic operations such as matrix multiplications and has been extensively used for privacy-preserving neural network inference and training [41, 12, 34, 36, 11]. Because secret-sharing offers information-theoretic security, computation can proceed on short integers, the bitlength of which does not depend on security parameters, aiding efficiency.

Traditionally performance of secret sharing techniques has been measured in terms of two parameters: the number of interactive operations and the number of sequential interactive operations, or rounds. However, for some computations such as matrix multiplication local operations can dominate the overall cost of the secure protocol. Traditional secret sharing techniques such as Shamir secret sharing [40] carry out computation on protected data over a field, most commonly set up as $\mathbb{Z}_p$ with prime p. This makes frequent use of modulo reduction a necessity, increasing the cost of the computation. To improve performance and directly utilize native instructions of modern processors, researchers turned to computation over ring $\mathbb{Z}_{2^k}$ [9, 5, 13, 16]. Unfortunately, Shamir secret sharing – a popular and efficient choice for computation in the honest majority setting – cannot be used for computation over $\mathbb{Z}_{2^k}$ and we must seek other options.

The honest majority setting, which assumes that only a minority of the parties carrying out the computation can be corrupt, offers great performance with reasonable trust assumptions, making a good performance-security trade-off. The techniques we are aware of in this setting which can perform computation over ring $\mathbb{Z}_{2^k}$ for some k are limited to a fixed number of parties, most commonly to 3 (see, e.g., [5, 28, 34, 12, 11]). This means that the techniques cannot generalize to any other number of participants, should there be a need to change the computation setup. This is the task we set to address in this work and generalize computation based on replicated secret sharing to support more than $n = 3$ computational parties.

Our contributions. Our contributions can be summarized as follows:

- We mimic the set of elementary building blocks used for constructing efficient protocols for Shamir secret sharing and develop their equivalents for replicated secret sharing defined over an arbitrary ring in the semi-honest setting. These building blocks include generating shares of pseudorandom integers and ring elements, multiplication, reconstructing a value from shares and multiplication followed by reconstruction as a single building block, denoted by MulPub. We optimize the solutions to lower communication complexity by relying on a pseudo-random function. This means that the techniques are computationally secure and they also come with formal security proofs. Our solutions are efficient and, for example, the cost of multiplication when instantiated with three parties matches the best known custom results which apply to the three-party setting only [5, 41].
- We build on the techniques of [16] to develop higher-level protocols over $\mathbb{Z}_{2^k}$ such as for random bit generation, comparisons, conversion between different ring sizes and more to enable general-purpose computation in this framework.
- We provide extensive benchmarks to evaluate performance of the developed techniques and compare them to the state of the art. We observe that when $n = 3$ our techniques are several times faster than their field-based counterparts when a sufficient number of operations is executed in parallel (i.e., computation is not network-bound) and up to 100 times faster for operation such as matrix multiplication.
- We also evaluate performance of our techniques on machine learning applications, namely, neural network predictions and support vector machine classification. Our runtimes are also up to an order of magnitude faster than similar field-based implementation and perform similarly to recent custom machine learning protocol which can only support three parties.

Because our techniques are based on replicated secret sharing, it is expected that they will be used with a relatively small number of parties. This is similar to the fastest known secret sharing techniques based on Shamir secret sharing (e.g., [10, 8]) which also rely on replicated secret sharing for certain operations.

2 Related Work

Secret sharing [40, 7] is a popular choice for secure multi-party computation, and common options include Shamir secret sharing [40], additive secret sharing, and more recently replicated secret sharing [23] for three parties. Computation over rings, and specifically rings $\mathbb{Z}_{2^k}$, has recently gained attention, and publications that use this setting include [9, 5, 28, 13, 17, 16, 20, 2, 25]. We can distinguish between three-party techniques based on replicated secret sharing such as [9, 5, 28, 17, 20, 2, 25]; multi-party techniques based on additive secret sharing such as [13, 16], typically for the setting with no honest majority; and ad-hoc techniques for three or four parties that utilize one or more types of rings with constructions for specific applications such as [24] and others.

The first category is the closest to this work and includes publications such as Sharemind [9], a well-developed framework for three-party computation with a single corruption using custom protocols; Araki et al. [5] who use three-party with a single corruption to support arithmetic or Boolean circuits; and several compilers from passively secure to actively secure protocols [28, 17, 20, 2]. Dalskov et al. [15] also studies four-party computation with a single corruption. We are not aware of existing multi-party techniques with honest majority over a ring which extend beyond three parties or multi-party protocols based on replicated secret sharing over a ring. While replicated secret sharing is meaningful only for a small number of parties, we still find it desirable to support more participants and build additional techniques for this setting. For example, if our matrix multiplication protocol over a ring with three parties is 100 times faster than conventional field-based computation, it will remain faster even if the work increases when the number of parties is larger than 3.

We rely on the results of Damgard et al. [16] for some of our protocols. While this work is for the SPD$\mathbb{Z}_{2^k}$ framework [13] in the malicious setting with no honest majority, once we develop elementary building blocks, the structure of higher-level protocols can remain similar.

Literature on privacy-preserving machine learning is also related to this work. We distinguish between two-party solutions where one party holds the model and the other party holds the input on which the model is to be evaluated and between multi-party (typically, three-party) solutions. Publications from the first category include MiniONN [29] which studied neural network evaluation using secret sharing and homomorphic encryption; Gazelle [24] which combined homomorphic encryption with garbled circuits (GC) and additive secret sharing; DELPHI [33] which improved upon these techniques; and CrypTFlow2 [37] which also builds upon this work with a focus on deep neural networks. Chameleon [38] incorporates garbled circuits, the GMW protocol [32], and additive secret sharing.

Multi-party constructions provide protocols for training and prediction across multiple parties. SecureML [35] was one of the first publications to provide a two-server architecture for training neural networks (as wells as several other machine learning applications). ABY3 [34] combines techniques based on replicated and binary secret sharing with and GCs in the three-party setting with an honest majority. These techniques are further improved in Trident [12] and extended to the four-party setting. SecureNN [41] provides three- and four-party protocols for a variety of neural network functions under the same security assumption as ABY3. Their protocols are asymmetric, where parties have dedicated roles in a computation. ASTRA [11] is a three-party framework over the ring $\mathbb{Z}_{2^k}$ under a linear secret sharing scheme under both semi-honest and malicious security assumptions. Similar to SecureNN, protocols are asymmetric. They build upon this work with BLAZE [36]. Abspoel et al. [3] applies the MP-SPD$\mathbb{Z}$ [25] framework for secure outsourced training of decision trees. Their system operates under the three-party, honest-majority assumption with RSS.

3 Preliminaries

3.1 Secure Multi-Party Computation

We consider the conventional secure multi-party setting with n computational parties, out of which at most t can be corrupt. We work in the setting with honest majority, i.e., $t > n/2$ and focus is on the security against semi-honest participants and simulation-based security, formulated as follows:

Definition 1. *Let parties $P_1, \ldots, P_n$ engage in a protocol Π that computes function $f(\mathsf{in}_1, \ldots, \mathsf{in}_n) = (\mathsf{out}_1, \ldots, \mathsf{out}_n)$, where in_i and out_i denote the input and output of party P_i, respectively. Let*

$\text{VIEW}_\Pi(P_i)$ *denote the view of participant* P_i *during the execution of protocol* Π. *More precisely,* P_i*'s view is formed by its input and internal random coin tosses* r_i*, as well as messages* $m_1, \ldots, m_k$ *passed between the parties during protocol execution:* $\text{VIEW}_\Pi(P_i) = (\mathsf{in}_i, r_i, m_1, \ldots, m_k)$. *Let* $I = \{P_{i_1}, P_{i_2}, \ldots, P_{i_t}\}$ *denote a subset of the participants for* $t < n$, $\text{VIEW}_\Pi(I)$ *denote the combined view of participants in* I *during the execution of protocol* Π *(i.e., the union of the views of the participants in* I*), and* $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ *denote the projection of* $f(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ *on the coordinates in* I *(i.e.,* $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ *consists of the* i_1*th, ...,* i_t*th element that* $f(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ *outputs). We say that protocol* Π *is* t*-private in the presence of semi-honest adversaries if for each coalition of size at most* t *there exists a probabilistic polynomial time (PPT) simulator* S_I *such that* $\{S_I(\mathsf{in}_I, f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)), f(\mathsf{in}_1, \ldots, \mathsf{in}_n)\} \equiv \{\text{VIEW}_\Pi(I), (\mathsf{out}_1, \ldots, \mathsf{out}_n)\}$, *where* $\mathsf{in}_I = \bigcup_{P_i \in I} \{\mathsf{in}_i\}$ *and* $\equiv$ *denotes computational or statistical indistinguishability.*

As customary with techniques based on secret sharing, the set of computational parties does not have to coincide with (and can be formed independently from) the set of parties supplying inputs in the computation (input providers) and the set of parties receiving output of the computation (output recipients). Then if a computational party learns no output, the computation should not reveal any information to that party. Consequentially, if we wish to design a functionality that takes input in the secret-shared form and produces shares of the output, any computational party should learn nothing from protocol execution.

3.2 Secret Sharing

A secret sharing scheme allows one to produce shares of secret x such that access to a predefined number of shares reveals no information about x. In the context of secure multi-party computation, each of the n participants receives one or more shares x_i and in the case of (n, t) threshold secret sharing schemes, possession of shares stored at any t or fewer parties reveals no information about x, while access to shares stored at $t+1$ or more parties allows for reconstruction of x. Of particular importance to secure multi-party computation are linear secret sharing schemes, which have the property that a linear combination of secret shared values can be performed locally on the shares. For example, to add secret shared x and y, each participant can locally compute $x_i + y_i$ on shares it possesses. Examples of linear secret sharing schemes include additive secret sharing with $x = \sum_i x_i$ (as used in Sharemind [9] with $n = 3$ and in SPDZ [18] with any n), Shamir secret sharing which realizes (n, t) secret sharing with $t < n/2$ and represents a share as evaluation of a polynomial on a distinct point, and replicated secret sharing discussed next.

3.3 Replicated Secret Sharing

Our techniques utilize replicated secret sharing (RSS) [23] which has an associated access structure Γ. An access structure is defined by qualified sets $Q \in \Gamma$, which are the sets of participants who are granted access, and the remaining sets of the participants are called unqualified sets. In the context of this work we only consider threshold structures in which any set of t or fewer participants is not authorized to learn information about private values (i.e., they form unqualified sets), while any $t + 1$ or more participants are able to jointly reconstruct the secret (and thus form qualified sets). RSS can be defined for any $n \geq 2$ and any $t < n$. To secret-share private x using RSS, we treat x as an element of a finite ring $\mathcal{R}$ and additively split it into shares x_T such that $x = \sum_{T \in \mathcal{T}} x_T$ (in $\mathcal{R}$), where $\mathcal{T}$ consists of all maximal unqualified set of Γ (i.e., all sets of t parties in our case). Then each party $p \in [1, n]$ stores shares x_T for all $T \in \mathcal{T}$ subject to $p \notin T$. In the general case of (n, t)-threshold RSS, the total number of shares is $\binom{n}{t}$ with $\binom{n-1}{t}$ shares stored by each party,

which can become large as n and t grow. In what follows, we use notation $[x]$ to mean that (private) x is secret shared among the parties using RSS.

Example. In the $(4, 2)$ setting, $\mathcal{T}$ consists of 6 sets $\mathcal{T} = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$ and thus there are 6 corresponding shares for every secret-shared x. Then party 1 stores shares $x_{\{2,3\}}, x_{\{2,4\}}, x_{\{3,4\}}$, party 2 stores $x_{\{1,3\}}, x_{\{1,4\}}, x_{\{3,4\}}$, etc.

The parties will need to perform computation on secret shared values. The first important property of RSS is that it is linear. For example, to add $[a]$ and $[b]$, party p computes $a_T + b_T$ (in $\mathcal{R}$) for each $T \in \mathcal{T}$ that p stores. A number of other operations, such as multiplications, reconstructing a value from its shares, etc., are interactive. We consequentially describe in Section 4 the way we realize these operations for the purposes of this work. An important optimization on which we rely is non-interactive evaluation of a pseudo-random function (PRF) using RSS in the computational (as opposed to number-theoretic) setting as proposed in [14]. We detail our use of this functionality in Section 4.

In what follows, we use the notation $\leftarrow$ to denote output of randomized algorithms, while the notation $=$ refers to deterministic assignment.

4 Basic Protocols

Recall that RSS enjoys the linear property. For example, we can add secret-shared $[a]$ and $[b]$ by adding a_T and b_T for each $T \in \mathcal{T}$ that a party possesses. We also use the ability to add/subtract known integers to a secret-shared value $[a]$ and multiply a secret-shared value $[a]$ by a known integer. In the former case, we realize $[a] + b$ by converting b to $[b]$ and performing the addition, where shares b_T use no randomness (e.g., we could set one share to b and the remaining shares to 0 to maintain that $\sum_{T \in \mathcal{T}} b_T = b$). In the latter case, $c = [a] \cdot b$ is realized by setting $c_T = a_T \cdot b$ (in $\mathcal{R}$) for each $T \in \mathcal{T}$.

For convenience and without loss of generality, we let $n = 2t + 1$. In the case when $n > 2t + 1$, $2t + 1$ parties can carry out the computation on a reduced set of shares in such a way that there is no need to involve the remaining parties in the computation.

4.1 Random Number Generation

We will be using two types of random number generation, which we discuss here.

4.1.1 PRG

Invocation of $[a_1], [a_2], \ldots \leftarrow \mathsf{PRG}([s])$ is realized by independently executing a PRG algorithm on each share of s without interaction between the parties. Because the output of $\mathsf{PRG}([s])$ is private, we expect it to produce a sequence of secret-shared values (represented as ring elements). Furthermore, in our construction we only call the PRG to obtain random (secret-shared) ring elements. This means that calling $\mathsf{PRG}(s_T)$ to produce pseudo-random a_T will result in $\mathsf{PRG}([s])$ generating $[a]$, where a is pseudo-random as well because $a = \sum_{T \in \mathcal{T}} a_T$ (in $\mathcal{R}$). This is similar to evaluating a PRF on a secret-shared key in the RSS setting without interaction in [14].

$\mathsf{PRG}(s_T)$ can be realized internally using any suitable algorithm, as long as it is consistent among the computational parties. For example, because of the speed of AES encryption on modern processors, one might implement $\mathsf{PRG}(s_T) = \mathsf{PRF}(s_T, 0)||\mathsf{PRF}(s_T, 1)||\ldots$, where $\mathsf{PRF} : \mathcal{R} \times \{0, 1\}^\kappa \to \mathcal{R}$ is a PRF instantiated with AES.

Let $\mathsf{G} = \mathsf{PRG}([s])$. When the output of G is not consumed all at once, we use notation $\mathsf{G}.\mathsf{next}$ to retrieve the next (secret-shared) element from G. Similarly, if $\mathsf{G}_T = \mathsf{PRG}(s_T)$, notation $\mathsf{G}_T.\mathsf{next}$

refers to the next pseudo-random share output by G_T.

4.1.2 PRandR

$[a] \leftarrow \mathsf{PRandR}()$ computes a secret-shared random element of ring $\mathcal{R}$. We implement this function by executing $\mathsf{PRG}([k]).\mathsf{next}$, where k is a system-wide key. The key k is set up at the system initialization time (in the form of secret shares) and does not change throughout program execution.

4.2 Multiplication

Multiplication $[c] \leftarrow [a] \cdot [b]$ can be realized using the fact that $a = \sum_{T \in \mathcal{T}} a_T$ and $b = \sum_{T \in \mathcal{T}} b_T$ and thus $[a] \cdot [b] = \sum_{T_1, T_2 \in \mathcal{T}} a_{T_1} \cdot b_{T_2}$ (in $\mathcal{R}$). Note that for any (T_1, T_2) pair, there will be a party holding shares T_1 and T_2, and thus performing this operation involves local multiplication and addition over different choices of T_1, T_2. More formally, let mapping $\rho : \mathcal{T} \times \mathcal{T} \rightarrow [1, n]$ denote a function that for each pair $(T_1, T_2) \in \mathcal{T}^2$ dedicates a party $p \in [1, n]$ responsible for computing the product $a_{T_1} \cdot b_{T_2}$ (clearly, p must possess shares T_1 and T_2). For performance reasons, we also desire that ρ distributes the load among the parties as fairly as possible.

As a result of this (local) computation, the parties hold additive shares of the product $a \cdot b = c$, which needs to be converted to RSS for consecutive computation. This conversion was realized in early publications [31, 6] by having each party create replicated secret shares of their result and distribute each share to the parties entitled to knowing it (i.e., party p receives shares from each party for each $T \in \mathcal{T}$ subject to $p \notin \mathcal{T}$). This results in each participant creating $\binom{n}{t}$ shares and sending $\binom{n-1}{t}$ of them to each party. Consequentially, each participant adds the values received for share T and stores the sum as c_T, for each T in its possession.

More recent work such as [5] and others traded information-theoretic security (in the presence of secure channels) for communication efficiency by having the parties generate shared (pseudo-)random values. We pursue this direction in this work as well. However, if this idea is applied naively, it still results in unnecessarily high overhead. In particular, if we instruct each party p to choose $\binom{n}{t} - 1$ pseudo-random shares, compute the last share as a function of these pseudo-random shares and the computed value, and send the computed share to each of the parties entitled to having it, the inefficiency stems from the fact that p generates all possible shares, even those to which it is not entitled to have access. This results in many shares not contributing to secrecy.

To improve efficiency, our solution is to eliminate shares that do not contribute to secrecy and the corresponding key material for generating pseudo-random shares. Instead, our construction utilizes key material consistent with the setup of the RSS scheme. In particular, we use the same key setup as in PRSS, where k_T is known by all $p \notin T$. Then when a party needs to generate a pseudo-random share associated of its value for share T, the party will draw it from the PRG seeded with k_T.

We, however, note that multiple participants may need to draw from the PRG seeded with k_T to produce shares of their values, and it is generally not safe to use the same secret to protect multiple values, which is also the case in our application. Instead, multiple elements might be drawn from the PRG (seeded with k_T) to protect different values, and consistent use of the PRG with each seed can be setup by the participants ahead of time so that that information is public knowledge.

In addition to mapping ρ, our multiplication protocol uses another mapping $\chi : [1, n] \rightarrow \mathcal{T}$, which for each party p specifies the share T (subject to $p \notin T$) that p communicates (with all other shares of p's value being produced as pseudo-random elements). As before, we desire to choose the values of $\chi(p)$ as to evenly distribute the load and communication.

Protocol 1 $[c] \leftarrow [a] \cdot [b]$

// pre-distributed values are $[k]$ and public maps ρ and χ
// define $\mathsf{G}_T = \mathsf{PRG}(k_T)$

1: each $p \in [1, n]$ does the following
2: let $S_p = \{T \in \mathcal{T} \mid p \notin T\}$
3: $v^{(p)} = \sum_{T_1, T_2 \in \mathcal{T}, \rho(T_1, T_2) = p} a_{T_1} b_{T_2}$ (in $\mathcal{R}$)
4: **for** $T \in S_p$ **do**
5: $c_T = 0$
6: $v^{(p)}_{\chi(p)} = v^{(p)}$
7: **end for**
8: **for** $p' \in [1, n]$ in order **do**
9: **for** $T \in S_p$ **do**
10: **if** $(p' \neq p) \wedge (p' \notin T) \wedge (\chi(p') \neq T)$ **then**
11: $c_T = c_T + \mathsf{G}_T.\mathsf{next}$ (in $\mathcal{R}$)
12: **else if** $(p' = p) \wedge (\chi(p) \neq T)$ **then**
13: $z = \mathsf{G}_T.\mathsf{next}$
14: $c_T = c_T + z$ (in $\mathcal{R}$)
15: $v^{(p)}_{\chi(p)} = v^{(p)}_{\chi(p)} - z$ (in $\mathcal{R}$).
16: **end if**
17: **end for**
18: **end for**
19: send $v^{(p)}_{\chi(p)}$ to each $p' \notin \chi(p)$ (other than itself)
20: **for** $p' \in [1, n]$ such that $p \notin \chi(p')$ **do**
21: after receiving $v^{(p')}_{\chi(p')}$ from p', set $c_{\chi(p')} = c_{\chi(p')} + v^{(p')}_{\chi(p')}$ (in $\mathcal{R}$)
22: **end for**
23: $c_{\chi(p)} = c_{\chi(p)} + v^{(p)}_{\chi(p)}$ (in $\mathcal{R}$)
24: **return** $[c]$

The above intuition leads us to the optimized n-party multiplication protocol given as Protocol 1. After computing its private value $v^{(p)}$ according to ρ, each party p distributes it into $\binom{n-1}{t}$ additive shares (one of which is communicated while others are computed using PRGs). Afterwards, each party sets its c_T as a sum of $t+1$ shares (computed or received) of values $v^{(p')}$ for each party p' entitled to shares c_T. This matches the fact that each share a_T of secret a is maintained by $t+1$ parties. Correctness is achieved by ensuring that in Protocol 1 two different participants p and p' with access to shares T consistently associate the values that they draw from G_T with shares belonging to different parties by always processing the values in the increasing order of participants' IDs. Preparation of the shares in Protocol 1 is done on lines 10–16, where a participant either masks its share with a pseudo-random value because it is used by another party or forms its own shares and the value to be transmitted.

In this protocol, each party on average communicates t ring elements and draws $\binom{n-1}{t} - 1 + (n-1)\binom{n-2}{t} - t$ pseudo-random ring elements (which is equal to $(t+1)(\binom{n-1}{t} - 1)$ when $n = 2t+1$).[1] The latter can be explained by using $\binom{n-1}{t} - 1$ pseudo-random shares for its value being re-shared and $\binom{n-2}{t}$ shares that it has in common with any other party except the t values that it receives with a symmetric communication pattern. (Recall that each party maintains $\binom{n-1}{t}$ shares of a

[1] It is possible to distribute the load evenly among the parties by appropriately setting the χ function.

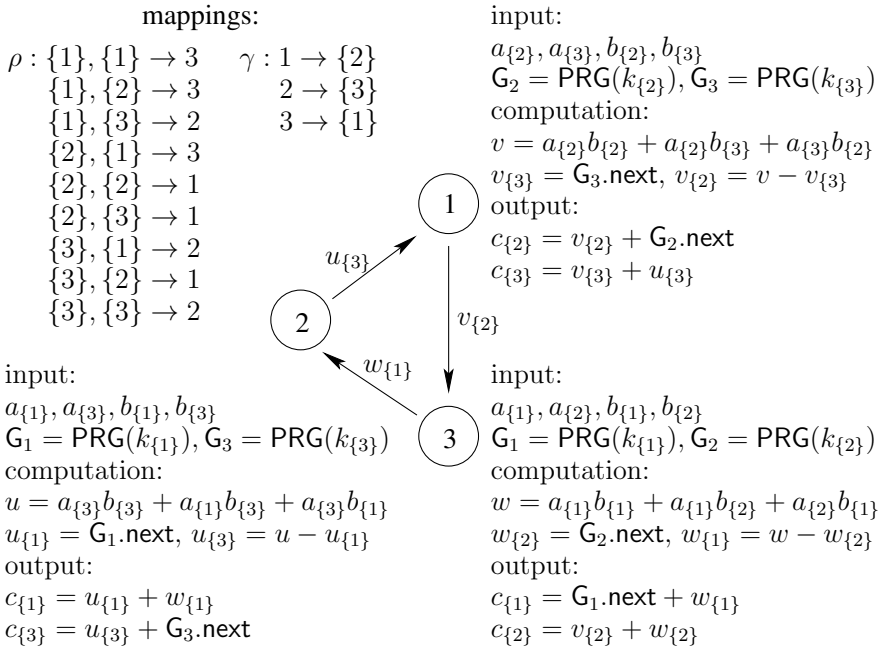


Figure 1: Illustration of three-party multiplication $[a] \cdot [b]$. All arithmetic is in $\mathcal{R}$.

secret and has $\binom{n-2}{t}$ shares in common with any other party). When the communication pattern is not symmetric, the overall amount of work and communication remains unchanged, but it may be distributed differently. Thus, we refer to the average work and communication in that case.

Compared to other multiplication protocols in the literature, the three-party version of our protocol matches communication of recent multiplication from [5], which (unlike this work) is available only for three parties. This also improves on communication of standard multiplication using Shamir secret sharing from [21] (information-theoretically secure in the presence of secure channels) by a factor of 2 and improves on communication of Sharemind's multiplication from [26] by a factor of 2.

Example. With three parties, we could have party 1 (in possession of shares $\{2\}$ and $\{3\}$) compute (and add) products $a_{\{2\}}b_{\{2\}}$, $a_{\{2\}}b_{\{3\}}$, and $a_{\{3\}}b_{\{2\}}$, party 2 (in possession of shares $\{1\}$ and $\{3\}$) compute products $a_{\{3\}}b_{\{3\}}$, $a_{\{1\}}b_{\{3\}}$, and $a_{\{3\}}b_{\{1\}}$, and party 3 (in possession of shares $\{1\}$ and $\{3\}$) compute products $a_{\{1\}}b_{\{1\}}$, $a_{\{1\}}b_{\{2\}}$, and $a_{\{2\}}b_{\{1\}}$. This defines mapping ρ. Also let $\chi(1) = \{2\}$, $\chi(2) = \{3\}$, and $\chi(3) = \{1\}$. This, for example, means that when party 1 divides its computed value $v^{(1)}$ into shares $v^{(1)}_{\{2\}}$ and $v^{(1)}_{\{3\}}$, the latter is computed using a PRG, while the former is being sent to party 3 (i.e., the other party entitled to have that share). An illustration of the multiplication protocol with these mappings in the three-party setting is given in Figure 1.

We state security of multiplication as follows:

Theorem 1. *Multiplication $[c] \leftarrow [a] \cdot [b]$ is secure in the (n, t) setting with $t = (n-1)/2$ according to definition 1.*

Proof. Let I denote the set of corrupt parties. We consider the maximal amount of corruption with $|I| = t$. Because the computation proceeds on secret shares and the parties do not learn the result, no information should be revealed to the computational parties as a result of protocol execution.

We build a simulator S_I that interacts with the parties in I as follows: when a party $p \in I$ expects to receive a value from another party $p' \notin I$ in step 5 of the computation according to

Operation	Rounds	$(3,1)$ setting		(n,t) setting	
		Comm	Crypto ops	Comm	Crypto ops
$\mathsf{PRG}([s]).\mathsf{next}$ and $\mathsf{PRandR}()$	0	0	2	0	$\binom{n-1}{t}$
$\mathsf{Mul}([a],[b])$	1	1	2	t	$(t+1)\left(\binom{n-1}{t}-1\right)$
$\mathsf{Open}([a])$	1	1	0	t	0
$\mathsf{MulPub}([a],[b])$	1	2	2	$n-1$	$t\binom{n-1}{t}$
$\mathsf{DotProd}(\langle[a^1],\ldots,[a^N]\rangle,\langle[b^1],\ldots,[b^N]\rangle)$	1	1	2	t	$(t+1)\left(\binom{n-1}{t}-1\right)$

Table 1: Performance of RSS operations.

function χ, S_I chooses a random element of $\mathcal{R}$ and sends it to p. S_I preserves consistency of the view and ensures that when the same value is to be sent by p' to multiple parties in I, all of them receive the same random value. This is the only portion of the protocol where corrupt parties can receive values (that the simulator produces), and the only portion of the protocol when a corrupt party p may send a value to an honest party p' is step 4, which S_I receives on behalf of p'. All other computation is local, in which S_I does not participate.

We next argue that the simulated view is computationally indistinguishable from the real view. First, note that the corrupt parties in I collectively hold shares a_T, b_T and keys k_T (and thus can compute values $\mathsf{G}_T.\mathsf{next}$) for each $T \in \mathcal{T}$ such that $\exists p \in I$ and $p \notin T$. This entitles the corrupt parties to computing the corresponding shares c_T, but the rest of the shares must remain unknown, so that they are unable to compute c. Next, notice that when $|I| = t$, there is only one share $T^* = I$ such that all parties $p \in I$ have no access to k_{T*} and c_{T^*}, while all parties $p' \notin I$ store those values. Then there are two cases to consider: (1) If one or more parties $p \in I$ receive $\chi(p')$'s share of $v^{p'}$ from another party $p' \notin I$ (it must be the case that $\chi(p') \neq T^*$), the received share has been masked by a fresh pseudo-random element from G_{T^*}, is therefore pseudo-random and indistinguishable from random by any $p \in I$. (2) If no party $p \in I$ receives a value from any given $p' \notin I$, indistinguishability is trivially maintained. □

The computation associated with multiplication can be generalized to compute the dot-product of two secret-shared vectors $\mathsf{DotProd}(\langle[a^1],\ldots,[a^N]\rangle,\langle[b^1],\ldots,[b^N]\rangle)$, or evaluate any other multivariate polynomial of degree 2, using the same communication and the same number of cryptographic operations as in multiplication. For that purpose, we only need to change the computation in step 3 of the multiplication protocol. For example, for $\mathsf{DotProd}$, we modify step 3 to compute $v^{(p)} = \sum_{T_1,T_2\in\mathcal{T},\rho(T_1,T_2)=p} \sum_{i=1}^{N} a^i_{T_1} b^i_{T_2}$ (in $\mathcal{R}$), while the rest of the steps remain unchanged.

4.3 Revealing Private Values

4.3.1 Open

Reconstruction of a secret shared value $a = \mathsf{Open}([a])$ amounts to communicating missing shares to each party so that the value could be reconstructed locally from all shares. Recall that there are $\binom{n}{t}$ total shares and each party holds $\binom{n-1}{t}$ of them. Thus, during this operation each party is to receive $d = \binom{n}{t} - \binom{n-1}{t}$ missing shares.

Our next observation is that when n is not small, the value of d will exceed n and transmitting d messages to each party is not needed. Because the value is reconstructed as the sum of all shares, it is sufficient to communicate sums of shares instead of the individual shares themselves. Recall that $[a]$ can be reconstructed by $t+1$ parties. This means that it is sufficient for a participant to receive one element (i.e., a sum of the necessary shares) from t other parties.

As before, we would like to balance the load between the parties and ideally have each party

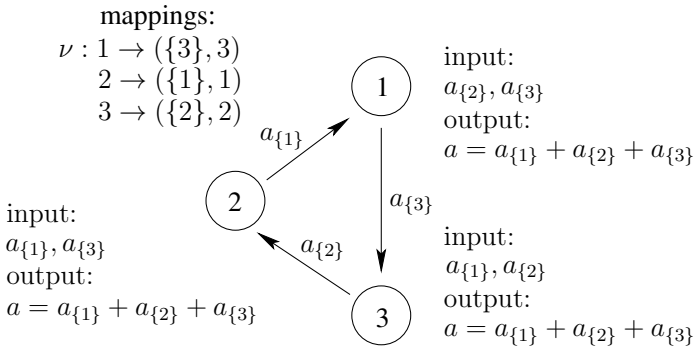


Figure 2: Illustration of three-party $\mathsf{Open}([a])$. All arithmetic is in $\mathcal{R}$.

transmit the same amount of data. This means that we instruct each party to send information to t other parties according to another agreed upon mapping $\nu : [1, n] \to (\mathcal{T}, [1, n])^d$. For each party p, this mapping will specify which of p's shares should be communicated to which other party. The mapping ν will then define computation associated with this operation: each p computes $\sum_{T,\nu(p)=T,p'} a_T$ (in $\mathcal{R}$) for each $p' \neq p$ present in the mapping and sends the result to p'.

Similar to other secret sharing frameworks, simply opening the shares of a maintains security of the computation (in the sense that no information about private values is revealed beyond the opened value a). This is because we maintain that at the end of each operation secret-shared values are represented using random shares. In particular, it is clear that the result of $\mathsf{PRG}([s]).\mathsf{next}$ and $\mathsf{PRandR}()$ produces random shares; shares are properly re-randomized during multiplication of $[a]$ and $[b]$, and shares of $[a] + [b]$ and $[a] - [b]$ are random if the shares of $[a]$ and $[b]$ are random themselves.

Example. With $n = 3$, we could have $\nu(1) = (\{3\}, 3)$, $\nu(2) = (\{1\}, 1)$, and $\nu(3) = (\{2\}, 2)$, which corresponds to $\nu(p) = (\{p-1\}, p-1)$ (where $p - 1 = 3$ for $p = 1$), which corresponds to the communication pattern in Figure 2.

4.3.2 MulPub

Functionality $c = \mathsf{MulPub}([a], [b])$ refers to multiplying two secret-shared $[a]$ and $[b]$ and opening their product c. It can be invoked multiple times in our Gen algorithm, e.g., on line 5, where the computation of $[s^{j,\overline{\alpha_j}}]$ involves multiplication. The reason why we are discussing this functionality is because in the past this operation could be implemented more efficiently than multiplication followed by an opening in alternative SS frameworks (e.g., see [10]), and we pursue a similar direction here. In the protocol we present here, MulPub is realized using a single round without increasing communication cost. Executing multiplication followed by Open would double the number of rounds.

In multiplication, after computing a product, each locally computed value is no longer random and needs to be re-randomized prior to opening it. In our RSS setting, we realize this by relying on pseudo-random values locally computed by the parties. In particular, similar to other building blocks, we associate a secret key k_T with each $T \in \mathcal{T}$ (i.e., this is the same key shares used with $\mathsf{PRandR}()$ and multiplication) and use pseudo-random values $\mathsf{G}_T.\mathsf{next}$ to protect the share of the product that each party locally computes, prior to that party revealing its randomized value to all others. To ensure that the product reconstructed by the parties is correct, we need to make sure that the sum of all blinding pseudo-random values equals to 0. In the three-party case, this can be accomplished by adding some pseudo-random values and subtracting others, as illustrated in

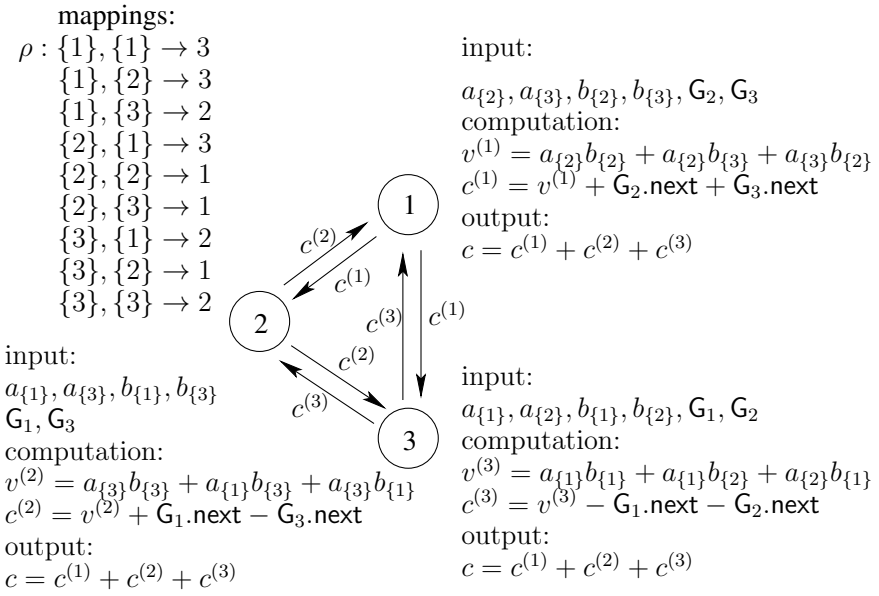


Figure 3: Illustration of three-party $\mathsf{MulPub}([a],[b])$. All arithmetic is in $\mathcal{R}$.

Figure 3.

With larger n and t we must be careful to draw new elements from each PRG to ensure that values released by different parties are protected using proper randomness without reusing them. This is similar to the logic used in multiplication. Then to realize this logic and ensure that all blinding factors add to 0, when multiple values are sampled from G_T, the last blinding value is set to the sum of all previously drawn elements multiplied by -1 (in $\mathcal{R}$). We provide a detailed description of MulPub in Protocol 2. G_T and S_p are defined as in multiplication.

In this algorithm, each party draws the same number of elements from each G_T in its possession to ensure that after single execution of this algorithm all parties are in the same state (by any given party may discard some of the computed values). Similar to the computation in multiplication, we order the parties based on the values of their IDs. Because any given share T is stored at $t+1$ parties, there are t calls to each G_T per invocation of this operation. Then the participant with the lowest ID among the parties with access to T ($j=0$) uses the first element of G_T to protect its value $v^{(p)}$ and disregards the $t-1$ other elements, the participant with the next lowest ID uses the second element, etc. The participant with the highest ID among those with access to T ($j=t$) computes the sum of all t elements drawn from G_T and subtracts the sum from its $v^{(p)}$. Correctness follows from the fact that the sum of all blinding values over all parties and all shares is equal to 0, i.e., $c = \sum_p c^{(p)} = \sum_p v^{(p)}$ (in $\mathcal{R}$).

To show security, we prove the following result:

Theorem 2. *The protocol $c = \mathsf{MulPub}([a],[b])$ is secure in the (n,t) setting with $t = (n-1)/2$ according to definition 1.*

Before proceeding with the proof, we demonstrate intuition behind it on the three-party example in Figure 3. Let z_T denote the output of $\mathsf{G}_T.\mathsf{next}$. Then party 1 transmits $c^{(1)} = v^{(1)} + z_{\{2\}} + z_{\{3\}}$, party 2 transmits $c^{(2)} = v^{(2)} + z_{\{1\}} - z_{\{3\}}$, and party 3 transmits $c^{(p)} = v^{(3)} - z_{\{1\}} - z_{\{2\}}$, where $c = v^{(1)} + v^{(2)} + v^{(3)}$ and each $v^{(i)}$ needs to be protected (all arithmetic is in $\mathcal{R}$). Without loss of generality, let party 3 be corrupt. Then party 3 (with access to $z_{\{1\}}$ and $z_{\{2\}}$) can compute

Protocol 2 $c = \mathsf{MulPub}([a], [b])$

// pre-distributed values are $[k]$ and public map ρ

each $p \in [1, n]$ does the following
  $v^{(p)} = c^{(p)} = \sum_{T_1, T_2 \in \mathcal{T}, \rho(T_1, T_2) = p} a_{T_1} b_{T_2}$ (in $\mathcal{R}$)
  for $T \in S_p$ **do**
    let j be the number of parties $p' < p$ such that $p' \notin T$
    for $i = 0$ to $t - 1$ **do**
      $z = \mathsf{G}_T.\mathsf{next}$
      if $j = t$ **then**
        $c^{(p)} = c^{(p)} - z$ (in $\mathcal{R}$)
      else if $i = j$ **then**
        $c^{(p)} = c^{(p)} + z$ (in $\mathcal{R}$)
      end if
    end for
  end for
  send $c^{(p)}$ to all other parties
  $c = c^{(p)}$
  for $i = 1$ to $n - 1$ **do**
    upon receiving $c^{(p')}$ from distinct p', set $c = c + c^{(p')}$ (in $\mathcal{R}$)
  end for
return c

$v^{(1)} + z_{\{3\}}$, $v^{(2)} - z_{\{3\}}$, and the output of the computation c, but no information about $v^{(1)}$ or $v^{(2)}$ (assuming security of the PRG) other than their sum $v^{(1)} + v^{(2)}$. The latter, however, is already computable by party 3 using the output c and its share $v^{(3)}$, which reveals no extra information about a and b beyond their product.

Proof. As before, let I denote the set of corrupt parties with $|I| = t$. We build a simulator S_I that interacts with the parties in I as follows: after S_I extracts shares a_T, b_T, k_T ($T \in \mathcal{T}$ such that $\exists p \in I$ and $p \notin T$) from the corrupt parties and receives the output c from the trusted party, S_I computes $v^{(p)}$ as prescribed by the protocol for each $p \in I$ and also their sum $v_I = \sum_{p \in I} v^{(p)}$ (in $\mathcal{R}$). S_I sets $v^{(p)}$ values for the remaining $n - t$ parties to random elements of $\mathcal{R}$ subject to $\sum_{p \notin I} v^{(p)} = c - v_I$ (in $\mathcal{R}$). S_I, acting on behalf of party $p \notin I$, sends the corresponding $v^{(p)}$ to each party in I.

To show that this simulation is indistinguishable from the real protocol execution, recall that there will be at least one T, denoted by $T^* = I$, to which the parties in I have no access (and thus correspondingly cannot distinguish the output G_{T^*} from random elements of the ring). During real protocol execution the parties in I receive $t + 1$ values $c^{(p)}$, one per $p \notin I$. With the knowledge that the corrupt parties collectively have, they can remove the effect of all randomization except the use of the output of G_{T^*}. If we let z_{i,T^*} denote the ith call to $\mathsf{G}_{T^*}.\mathsf{next}$ during the execution of MulPub in Protocol 2, then the corrupt parties can recover t values of the form $v^{(p)} + z_{i,T^*}$ with unique p and i and one value of the form $v^{(p)} - \sum_{i=1}^{t} z_{i,T^*}$ for another p. The next thing to notice is that any t (out of $t + 1$) of these values are pseudo-random and computationally protect the corresponding $v^{(p)}$ values. The introduction of the remaining value reveals the sum of all $v^{(p)}$s, but not other information (i.e., the last value corresponds to the difference to make the sum equal to $c - v_I$). This means that substituting these values with random elements subject to $\sum_{p \notin I} v^{(p)} = c - v_I$ provides the same information to the corrupt parties and achieves computational

Protocol 3 $[b] \leftarrow \mathsf{RandBit}()$

1: $[u]_{k+2} \leftarrow \mathsf{PRandR}(k+2)$;
2: $[a]_{k+2} = 2[u]_{k+2} + 1$;
3: $e = \mathsf{MulPub}([a]_{k+2}, [a]_{k+2})$;
4: compute the smallest root of e modulo 2^{k+2} and denote it by c; compute the inverse of c modulo 2^{k+2} and denote it by c^{-1};
5: $[d]_{k+2} = c^{-1}[a]_{k+2} + 1$;
6: for each $T \in \mathcal{T}$, let share $b_T = d_T/2$;
7: **return** k least significant bits of each b_T as $[b]_k$;

indistinguishability of the views. □

Similar to multiplication, MulPub can be generalized to evaluate any (multi-variate) polynomial of degree 2 and open the result. For example, in Gen we compute and open $s^{j,\overline{\alpha_j}} = s^{j,1} + \alpha_j(s^{j,0} - s^{j,1})$ using this operation.

5 Composite Protocols

While the previous operations can be instantiated to work with any finite ring, the techniques in this section work only in a ring $\mathbb{Z}_{2^k}$ for some k. Ring $\mathbb{Z}_{2^k}$ (for an appropriate choice of k) is the primary reason for supporting secure computation over rings because it allows us to utilize native CPU instructions for ring operations.

Because the goal of this work is to enable efficient general-purpose computation over rings $\mathbb{Z}_{2^k}$, we focus on major building blocks which can be consequentially used to compose a protocol for arbitrary functionalities including machine learning tasks. Of central importance to this effort is the development of comparison protocols (for both less-than comparison and equality testing functionalities), which are known to be difficult to design in a framework where the elementary techniques are based on arithmetic gates. Others include bit decomposition and truncation (i.e., division by a known power of 2). Combined together, these techniques can enable Boolean, integer, fixed-point, and even floating-point arithmetic as well as array and related operations, giving the ability to compose general-purpose protocols.

5.1 Random Bit Generation

Random bit generation, denoted RandBit, is a crucial building block used in a large variety of protocols including different types of comparisons, bit decomposition, division, etc. Therefore, it is of central importance to support this functionality for general-purpose computation. Our starting point for RandBit over a ring $\mathbb{Z}_{2^k}$ was the random bit generation protocol from [16] which modified field-based random bit generation from [10] to work in a ring $\mathbb{Z}_{2^k}$. We use the logic of [16] and adjust the algorithm to work in our setting. The resulting protocol is shown as Protocol 3.

To achieve 50% probability of each outcome of this operation, the computation has to proceed over a larger ring $\mathbb{Z}_{2^{k+2}}$ for most steps of the RandBit protocol, where the original computation was set up over ring $\mathbb{Z}_{2^k}$. Consequentially, we use notation $[x]_\ell$ with variable ℓ to denote that shares and computation are over ring $\mathbb{Z}_{2^\ell}$. We also parameterize function PRandR by the desired bitlength and $\mathsf{PRandR}(\ell)$ denotes that the function returns a random ring element from $\mathbb{Z}_{2^\ell}$.

Correctness of our construction follows from [16] and security also follows from the logic. That is, because we only disclose a random value e in the protocol and otherwise use secure building

Protocol 4 $[a_{k-1}] \leftarrow \mathsf{MSB}([a])$, where $a = \sum_{i=0}^{k-1} a_i 2^i \in \mathbb{Z}_{2^k}$

1: $[b], [r_0], \ldots, [r_{k-1}] \leftarrow \mathsf{RandBit}()$;
2: $[r'] = \sum_{i=0}^{k-2} [r_i] 2^i$;
3: $[r] = \sum_{i=0}^{k-1} [r_i] 2^i$;
4: $c = \mathsf{Open}([a] + [r])$;
5: $c' = c \bmod 2^{k-1}$;
6: for $i = 0$ to $k-2$ in parallel $[r_i]_1 \leftarrow \mathsf{A2B}([r_i])$;
7: $[u]_1 \leftarrow \mathsf{BitLT}(c', [r_0]_1, \ldots, [r_{k-2}]_1)$;
8: $[a'] = c' - [r'] + 2^{k-1}[u]_1$;
9: $[d] = [a] - [a']$;
10: $e = \mathsf{Open}([d] + 2^{k-1}[b])$ and let e_{k-1} be the most significant bit of e;
11: $[a_{k-1}] = e_{k-1} + [b] - 2e_{k-1}[b]$;
12: **return** $[a_{k-1}]$;

blocks, no information about private values can be leaked. The protocol runs in a single round using the same communication as MulPub over $\mathbb{Z}_{2^{k+2}}$. To improve performance, in our implementation the square root and inverse operations are carried out simultaneously in step 4.

5.2 Comparisons

Less-than (or greater-than) comparisons and equality tests can also be realized in this setting, using the results of [16]. We briefly describe the comparison protocol here. Traditionally, to compute $a < b$, we first compute the difference $a - b$ and then compute the most significant bit of the difference, i.e., negative or positive, which informs the result. The protocol for out setting adapted from MSB from [16] is given as Protocol 4.

It relies on two additional building blocks, namely arithmetic to binary conversion A2B and comparisons on bit-decomposed values BitLT from [39]. A2B locally converts shares over $\mathbb{Z}_k$ into shares over $\mathbb{Z}_2$ so that BitLT (which works with bits) could be implemented more efficiently.

Security of the algorithm follows from prior work and the fact that we use a composition of secure building blocks. In particular, the only values revealed in the protocol (in steps 4 and 10) are information-theoretically protected using freshly generated randomness. The round complexity of this protocol is $\log(k-1) + 3$ (where $\log(k-1)$ is due to BitLT) and the first round can be precomputed. Communication involves sending elements of $\mathbb{Z}_{2^k}$, $\mathbb{Z}_{2^{k+2}}$, and $\mathbb{Z}_2$. If we convert the communication to the number of elements in $\mathbb{Z}_{2^{k+2}}$ and use the cost of the multiplication protocol as a unit of interactive operation, this protocol has communication cost $< 2k + 6$.

To correctly implement the comparison operation of two k-bit integers over ring $\mathbb{Z}_{2^k}$, one would need to invoke the MSB protocol 3 times. However, correctness is also guaranteed if we compare two $(k-1)$-bit integers over ring $\mathbb{Z}_{2^k}$ using a single call to MSB. We use the latter approach in our implementation of machine learning algorithms.

There are noteworthy differences in the design of protocols developed for a ring as opposed to original protocols for a field. Certain operations such as prefix multiplication are not available in a ring and we resort to logarithmic round building blocks when protocols over a field achieve constant round complexity. In the context of comparison, a typical tool for realizing them was truncation (i.e., right shift), the cost of which was linear in the number of bits truncated, but the modulus had to be increased by a statistical security analysis to support such operations. In a ring, on the other hand, there is no significant increase in the ring size, but the communication cost is linear in

Protocol 5 $[a]_{k'} \leftarrow \mathsf{Convert}([a]_k, k, k')$, where $k' > k$

1: $[r_0]_k, \ldots, [r_{k-1}]_k \leftarrow \mathsf{RandBit}()$;
2: $[r]_k = \sum_{i=0}^{k-1} [r_i]_k 2^i$;
3: $c = \mathsf{Open}([a]_k - [r]_k)$;
4: for $i = 0$ to $k-1$ in parallel $[r_i]_1 \leftarrow \mathsf{A2B}([r_i]_k)$;
5: $[a_0]_1, \ldots, [a_{k-1}]_1 \leftarrow \mathsf{AddBitwise}(c, [r_0]_1, \ldots, [r_{k-1}]_1)$;
6: for $i = 0$ to $k-1$ in parallel $[a_i]_{k'} \leftarrow \mathsf{B2A}([a_i]_1, k')$;
7: **return** $[a]_{k'} = \sum_{i=0}^{k-1} [a_i]_{k'} 2^i$;

Protocol	Rounds	Communication
$\mathsf{RandBit}()$	1	2
$\mathsf{MSB}([a])$	$\log(k-1)+3$	$2k+6$
$\mathsf{Convert}([a]_k, k, k')$	$\log k + 4$	$2(k+k') + \log k + 7$

Table 2: Performance of composite protocols with communication measured in elementary interactive operations over $\mathbb{Z}_{2^{k+2}}$.

the bitlength of the ring and not in the bitlength of the truncated portion. This brings different trade-offs, but the availability of faster arithmetic in a ring will still lead to significant savings.

5.3 Share Conversion

In certain applications, such as neural network evaluation implemented over integers, the bitlength of the intermediate results grows during the computation. Combined with the fact the computation over shorter integers is of larger size, it would be beneficial to initially perform computation using a smaller ring size and consequentially convert it to a larger ring. For that reason, we discuss a procedure for converting a secret-shared value $[a]_k$ over $\mathbb{Z}_{2^k}$ to a different representation $[a]_{k'}$ over $\mathbb{Z}_{2^{k'}}$ for $k' > k$. We know how to do conversion between different field sizes in certain cases [19].

If we simply cast k-bit shares to k'-bit shares for $k' > k$, the overflow due to share addition in the absence of reduction modulo 2^k will affect the reconstructed value modulo $2^{k'}$. Thus, the task is to leave k least significant bits of a secret shared value and erase the remaining bits in a longer share representation. One way to achieve this is to invoke a truncation protocol $[a] \gg \ell$, e.g., as $([a] \cdot 2^{k'-k}) \gg 2^{k'-k}$ or $[a] - ([a] \gg k)2^k$. However, because computing precise truncation over a ring is costlier compared to the field algorithm, we design a more efficient version based on bit decomposition. In particular, we perform bit decomposition of $[a]_k$ into shares of bits in $\mathbb{Z}_2$, convert the bit shares to $\mathbb{Z}_{2^{k'}}$, and reassemble $[a]_{k'}$. This procedure is denoted by $\mathsf{Convert}$ and given as Protocol 5. It is based on the bit decomposition from [16] and invokes two additional sub-protocols: $\mathsf{B2A}$, which converts shares of a bit from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{k'}}$ and $\mathsf{AddBitwise}$, which adds two integers in a bit-decomposed form.

Security of Protocol 5 follows from the same argument as before: it invokes only secure building blocks and the disclosed value c is independent of private inputs. The round complexity is $\log k + 4$ and communication consists of elements of $\mathbb{Z}_{2^{k+2}}$, $\mathbb{Z}_{2^k}$, $\mathbb{Z}_2$, and $\mathbb{Z}_{2^{k'+2}}$. When converted to elementary interactive operations, we obtain $< 2(k+k') + \log k + 7$. A summary of the protocols discussed in this section is given in Table 2.

	Protocol	Batch Size						
		1	10	10^2	10^3	10^4	10^5	10^6
LAN	Shamir [43] (30 bits)	0.119	0.162	0.484	1.82	17.4	190	1,978
	Shamir [43] (60 bits)	0.121	0.164	0.509	1.94	20.3	208	2,190
	Shamir [8] (30 bits)	0.118	0.143	0.436	1.39	13.5	141	1,428
	Shamir [8] (60 bits)	0.121	0.145	0.459	1.51	15.2	160	1,639
	RSS (30 bits)	0.114	0.116	0.129	0.33	2.51	19.9	158
	RSS (60 bits)	0.109	0.117	0.144	0.495	4.31	30.0	268
WAN	Shamir [43] (30 bits)	20.9	20.9	21.3	25.3	51.0	529	4,637
	Shamir [43] (60 bits)	20.9	20.9	21.4	25.5	128	948	5,854
	Shamir [8] (30 bits)	14.0	14.1	14.3	17.3	34.7	304	2,526
	Shamir [8] (60 bits)	14.0	14.1	14.4	17.6	51.3	557	4,152
	RSS (30 bits)	13.9	14.0	14.0	14.2	16.8	172	1,817
	RSS (60 bits)	14.0	14.0	14.0	14.4	32.5	277	3,389

Table 3: Performance of multiplication protocols in milliseconds.

	Protocol	Matrix Size			
		10×10	100×100	500×500	1000×1000
LAN	Shamir [43] (30 bits)	0.446	178	19,600	$1.69 \cdot 10^6$
	Shamir [43] (60 bits)	0.467	184	20,172	$1.72 \cdot 10^6$
	RSS (30 bits)	0.151	9.70	1,230	12,200
	RSS (60 bits)	0.165	11.0	1,460	14,897
WAN	Shamir [43] (30 bits)	21.8	285	20,610	killed
	Shamir [43] (60 bits)	21.8	368	21,085	killed
	RSS (30 bits)	14.0	32	2,347	29,652
	RSS (60 bits)	14.0	49.9	2,862	30,131

Table 4: Performance of matrix multiplication in milliseconds.

6 Performance Evaluation

We implemented the protocols described in this work and evaluate their performance. We run both micro-benchmarks to evaluate the individual protocols as well as offer evaluation of machine learning applications.

The implementation was done in C++. We use AES from the OpenSSL cryptographic library [1] to instantiate the PRF and also to implement secure communication channels between each pair of the computational parties. We report the average execution time of 1000 executions for the micro-benchmark experiments and the average time of 5 executions for the application experiments. The runtimes are also averaged across the computational parties.

All experiments were run in a three-party setting. For LAN experiments, we used three 8-core 2.1GHz machines with 64GB of RAM. They were connected via a 1 Gbps Ethernet link with one-way latency of 0.15ms. For WAN benchmarks, we used two of the machines above and one remote 2.4HGz machine. One-way latency between the remote and local machines was 23ms. Although the machine configurations are slightly different, this should not introduce inconsistencies in the experiments because our protocols are interactive and symmetric, and the computation time depends on the communication links and performance of the slowest machine. All experiments use a single core.

	Protocol	Batch Size					
		1	10	10^2	10^3	10^4	10^5
LAN	Shamir [43] (30 bits)	0.178	0.396	1.08	9.71	95.3	986
	Shamir [43] (60 bits)	0.199	0.455	1.31	12.1	119	1,241
	RSS (30 bits)	0.103	0.120	0.213	0.515	4.47	54.0
	RSS (60 bits)	0.104	0.133	0.296	0.897	11.2	98.6
WAN	Shamir [43] (30 bits)	21.4	21.6	23.6	35.9	160	1,591
	Shamir [43] (60 bits)	21.4	21.6	24.1	38.7	178	1,935
	RSS (30 bits)	20.9	20.9	21.0	21.9	30.4	301
	RSS (60 bits)	20.9	20.9	21.9	22.8	143	853

Table 5: Performance of RandBit protocols in milliseconds.

	Protocol	Batch Size					
		1	10	10^2	10^3	10^4	10^5
LAN	Shamir [43] (30 bits)	3.11	5.07	39.8	394	4,064	40,339
	Shamir [43] (60 bits)	4.01	11.0	96.2	947	9,614	96,019
	RSS (30 bits)	0.842	1.233	4.826	20.65	192.4	1,873
	RSS (60 bits)	1.032	2.256	12.20	73.69	627.1	6,273
WAN	Shamir [43] (30 bits)	148	156	215	1086	7,364	67,497
	Shamir [43] (60 bits)	151	166	412	2,198	18,223	175,995
	RSS (30 bits)	125	126	129	199	909	8,059
	RSS (60 bits)	125	127	139	497	3,034	34,041

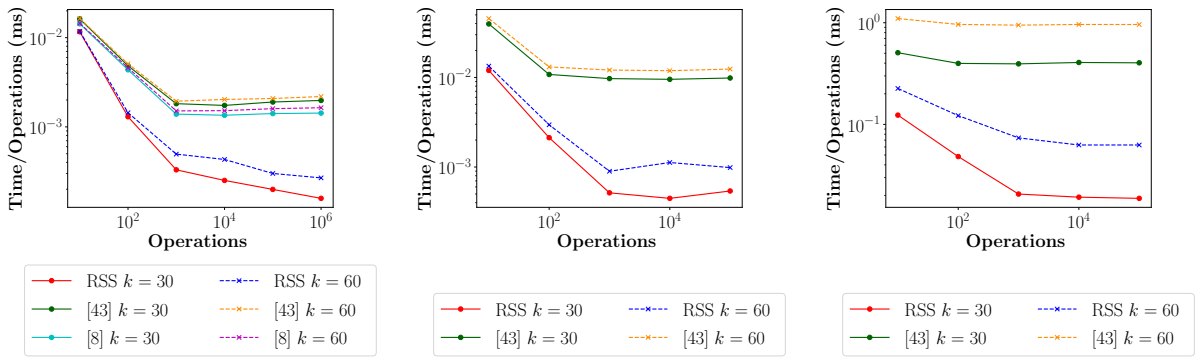
Table 6: Performance of most significant bit MSB protocols in milliseconds.

6.1 Micro-benchmarks

In this section we report on the results of our micro-benchmark experiments for the following operations: multiplication, matrix multiplication, random bit generation (RandBit) and comparison (MSB). The experiments were set up to use two bitlengths: $k = 30$ and $k = 60$. Internally, this allows us to use the `unsigned` and `unsigned long` integer types, respectively, to implement ring operations.

Tables 3 and 4 report on performance of multiplication and matrix multiplication operations, respectively. Because the most meaningful comparison of our results is to those in a comparable setting – i.e., semi-honest security with honest majority – over a field, we report performance of these protocols using implementation of Shamir secret sharing from PICCO [43]. Also, because [8] recently improved the multiplication protocol that [43] uses, we additionally include the results of evaluating the multiplication protocol from [8]. Since the techniques from [43] and [8] perform computation over a field, in order to represent all k-bit integers (for $k = 30$ and $k = 60$ in our experiments), we use field $\mathbb{Z}_p$ with a $(k+1)$-bit prime p. Batch size in the tables denotes how many operations were executed at the same time in a single batch.

Interestingly, we note from Table 3 that RSS-based realization of multiplication is up to 12.5 times faster on the LAN for a sufficiently large number of parallel invocations than the multiplication protocol used in PICCO. Furthermore, the communication of the multiplication protocol from [8] matches the communication of our multiplication for 3 parties (i.e., one ring or field element transmitted per party), but our performance is still 9 times faster for large batch sizes than the performance of [8], despite the fact that with RSS each party has to perform local computation twice (once for each share). This informs us that the performance advantage must stem from the use of faster arithmetic and avoiding repeated use of modulo reduction. In the WAN setting,



(a) Mult($[a], [b]$) (b) RandBit() (c) MSB($[a]$)

Figure 4: Microbenchmarks results.

performance is heavily dominated by the network latency for batches of smaller sizes, but we see that our techniques can still be up to 3.9 times faster than those in [43] and up to 2.1 times faster than those in [8] for larger batch sizes.

Table 4 reports on matrix multiplication results, where we multiply two matrices of the same size specified in the table by simultaneously invoking the necessary number of dot-product protocols. We can see from the table that ring-based performance can be more than 100 times faster for large matrices compared to PICCO's field-based computation. This is expected because matrix multiplication performance is dominated by local work for matrices of large size (i.e., to multiply two $m \times m$ matrices, the local work is $O(m^3)$, while communication is $O(m^2)$). This is where the use of ring-based computation over $\mathbb{Z}_{2^k}$ is most beneficial and directly represents performance improvement due to faster underlying arithmetic. Furthermore, performance improvement is also well-pronounced for matrices of small size, as we have a 3-fold performance improvement for matrices as small as 10×10 on the LAN.

Tables 5 and 6 provide results of our random bit generation RandBit and most significant bit MSB protocols. To support k-bit computation, both protocols require that we use ring $\mathbb{Z}_{2^{k+2}}$ during a portion of the computation (while the remaining portion uses $\mathbb{Z}_{2^k}$. The equivalent protocols for Shamir secret sharing – namely, RRandBit and LTZ from [10] – require statistical hiding and therefore increase the modulus size by a security parameter κ. PICCO sets $\kappa = 48$ and therefore our Shamir secret sharing experiments use prime modulus of size 79 and 109 bits for $k = 30$ and $k = 60$ settings, respectively.

RandBit is a relatively simple protocol and both ours and the protocol from [10] implemented in PICCO communicate the same number of ring or field elements in the 3-party setting. Table 5, however, demonstrates that our ring-based solution is faster and the difference increases as we increase the batch size, reaching up to 21-fold improvement for large batch sizes. This indicates that local computation of field-based RandBit is its bottleneck. In particular, the square root computation in RandBit from [10] is implemented as a modulo exponentiation. While the modulus size is not very large, the cost of that operation still dominates performance of field-based RandBit.

The gap between PICCO's LTZ and our MSB is even slightly larger than the gap of RandBit protocols, as evidenced by Table 6. Both field- and ring-based implementations make a linear in k number of calls to RandBit, which contributes to the difference, but our implementation also executes BitLT over $\mathbb{Z}_2$, while PICCO's implementation uses a fixed field for all operations. As a result, our MSB is up to 21.5 times faster for large batch sizes.

It is also informative to compare our field vs. ring results with those of SPDZ. While SPDZ [18]

	MiniONN [29]	Gazelle [24]	SecureNN [41]	Shamir [8, 43]	RSS				
Batch Size	1	1	1	1	1	5	10	50	100
Latency	3580 + 5740	810	220	2581	341	313	303	304	309

Table 7: Performance of MNIST neural network prediction in milliseconds.

and its ring version SPD$\mathbb{Z}_{2^k}$ [13, 16] use a different adversarial model and different underlying techniques (although also based on secret sharing), we would like to know whether similar savings are achievable in different settings. Recalls that our composite protocols were influences by those developed for SPD$\mathbb{Z}_{2^k}$ and thus share a similar structure. [16] reports that performance of multiplication improved by a factor of 4.6–4.9 for multiplication and by a factor of 5.2–6.0 for MSB-based comparison on a 1Gbps LAN. The results are only provided as throughput improvement and do not report different batch sizes. In our experiments we observed greater improvements, up to 9 times for multiplication and up to 21.5 improvement for MSB. This may be explained by the fact that our techniques are more lightweight and perhaps switching to faster arithmetic make less of an impact in the SPDZ setting.

The last results that would like to mention is depicting timings per operation when executing a number of operations in parallel. Figure 4 provides these results. As the figures suggest, Shamir secret sharing-based multiplication protocols reach the bottleneck of computation at a batch size of 10^3, while our multiplication could benefit from parallel operation until the batch size reaches 10^6. We believe this is because the computation components in our protocol are fast, and the communication dominates the cost of entire procedure. This explains why the gap between our protocol and the others in WAN is smaller than the gap in LAN. Similar phenomena are also observed in benchmarks of MSB and RandBit. We can also see in Figure 4c that PICCO's MSB barely benefits from the parallel operation, while our protocol achieves a 40 times speed-up by increasing the batch size to 10^4. This illustrates that our protocol has a better potential to facilitate applications that requires batch operations.

6.2 Machine Learning Applications

We also evaluate our protocols on machine learning applications and show that they exhibit good performance. We consider neural network inference and support vector machine (SVM) evaluation, in part to facilitate comparison to prior work.

6.2.1 Neural Networks

A *neural network* is a series of interconnected layers consisting of neurons. Each neuron has an associated weight and bias that is used for computation on some input data, and outputs a prediction based on the data. There is a large variety of different types of neural networks, and for our benchmarking we chose the neural network from MiniONN [29] for the MNIST dataset [27] (Figure 12 in [29]) because it is a popular choice for evaluating privacy-preserving neural network inference.

The MNIST neural network evaluation (also given as Protocol 6 in Appendix A) uses convolution and fully-connected layers, which are implemented using matrix multiplication. It also uses Rectified Linear Units (ReLU) as the activation function, which for a vector $\mathbf{y}$ is defined as $f(\mathbf{y}) = [\max(0, y_i)]$. Lastly, it invokes max pooling of a window 2×2, which computes the maximum value of the elements in that window. ReLU and max pooling operations are implemented using MSB and multiplication operations and are designed to minimize the number of rounds (i.e.,

Protocol	Integer size	Batch Size				
		1	5	10	50	100
RSS	30	29.8	14.4	13.1	10.4	10.1
	60	58.9	39.0	35.3	33.9	30.9
SPD$\mathbb{Z}_{2^k}$ [16]	32	242 + 3055	162 + 3055	–	–	–
	64	362 + 10006	244 + 10006	–	–	–
Shamir [8, 43]	30	311	313	–	–	–
	60	705	689	–	–	–

Table 8: Performance of ALOI SVM classification in milliseconds.

computation of maximum uses hierarchical structure).

We use MiniONN's implementation choices and, in particular, run the computation on integer inputs. Because the inputs were originally floating-point values, they are scaled by a factor of 1000 and rounded to the nearest integer, including both positive and negative values. Because the bitlength of the intermediate results grows with each multiplication, [29]'s implementation ran the computation using a 37-bit modulus and avoided the use of truncation. However, we determined that this modulus is too small and 49 bits are necessary to correctly evaluate the model, which we consequentially use. Our implementation achieves the same 99.0% precision as reported for this model in [30] (corrected version of [29]).

While it is possible to perform the entire computation in $\mathbb{Z}_{2^{49}}$, we observe that the initial steps are of the largest size and uses significantly shorter integers than 49 bits. Because the cost of comparisons is linear in the bitlength of the ring elements, we can improve performance by several times if we start computation on shorter values and convert the intermediate results to a larger ring prior to multiplication that increases the size of the intermediate results. Therefore, we start computation with 20-bit integers and increase the ring size by 10 bits at different points of the computation as illustrated in Protocol 6.

The results of our experiments for the MNIST neural network model with three computational parties are presented in Table 7. In addition to listing MiniONN's performance (online and offline), we also include performance of Gazelle's two party construction [24] and SecureNN's custom 3-party solution from [41] for the same model. Lastly, we also provide performance of PICCO's program for the same functionality with improved multiplication from [8]. In order to do as close of a comparison as possible, our PICCO program is written to compute on integers of increasing sizes, which means that the cost of comparisons is linear in 20 in the first round of ReLU and max pooling operations just like in our ring-based implementation and gradually increases throughout the computation. PICCO determines the optimal modulus size for the computation, which in this case was computed to be 89 bits, and runs the entire computation using the same field size, but the cost of comparison operations is aligned with our ring-based implementation.

MiniONN represents somewhat outdated now performance in the two-party setting and our solution is significantly faster; similarly, it is about 2.4 times faster than Gazelle's performance. PICCO's field-based computation with three computational parties is 7.5 times slower, which is in line with the micro-benchmark results. Our solution also performs on par with custom 3-party SecureNN's solution. In particular, SecureNN builds protocols for PPML tasks using custom arithmetic for three parties, runs the experiments using more powerful hardware (specifically, Amazon EC2 c4.8xlarge instances), and does not specify how multi-threading may have been used. We consider achieving similar performance to be a significant achievement for a general-purpose framework that supports a varying number of participants.

Several other publications benchmarked neural network predictions [34, 38, 4, 12, 11, 36, 33,

37]. However, because they either do not support or do not run the computation associated with MiniONN's MNIST neural network evaluation, we cannot directly compare our performance. For example, while ABY^3 [34] is said to use MiniONN's MNIST neural network, their experiments are actually based on a different, simpler model used in Chameleon [38].

6.2.2 Support Vector Machines

A *support vector machine* (SVM) is a type of a supervised learning classifier, where the computation is parameterized by the number of classes q and features m. We choose to do SVM classification and specifically run the computation for the ALOI dataset [22] to be able to compare performance of our framework to that of SPD$\mathbb{Z}_{2^k}$ reported in [16]. In particular, SPD$\mathbb{Z}_{2^k}$ achieves security in the malicious model with no honest majority over a ring – a much stronger security model than ours – and we are interested in knowing the computational price of the differences in the security assumptions.

SVM computation consists of a series of dot products of the feature vector and support matrix, all of which can be computed at the same time, followed by argmax computation of the resulting values. This computation is given as Protocol 7 in Appendix A, where argmax (similar to max) can be computed in a hierarchical manner. The results of our experiments for the ALOI dataset with 463 classes and 128 features are presented in Table 8.

We see that SPD$\mathbb{Z}_{2^k}$'s performance (combined offline and online) in the two-party setting is about 200 times slower due to their significantly stronger security model. We also see that PICCO's implementation is an order of magnitude slower and the times do not reduce as significantly with the increased batch size as in our ring-based setting. This tells us that a single invocation of SVM evaluation is network-bound over a ring, but this is not the case with field-based computation.

7 Conclusions

In this work we study multi-party threshold secret sharing over a ring in the semi-honest model with honest majority with the goal of improving performance compared to its field-based counterparts. We design low-level operations for n-party replicated secret sharing over an arbitrary ring and consequentially build on them to enable general-purpose protocols in this setting over ring $\mathbb{Z}_{2^k}$ for some k. Our implementation results demonstrate that ring-based implementations of different operations are several times faster than their field-based equivalents with $n = 3$. This allows us to improve performance of different applications including privacy-preserving machine learning tasks. We specifically test performance of neural network and support vector machine classification and find that performance of our techniques is on par with the best custom three-party protocols for those functions.

Acknowledgments

The authors would like to thank Jian Liu for help with understanding and reproducing the computation associated with the MNIST neural network evaluation in MiniONN [29]. The authors also acknowledge support from the Emulab project [42] for utilizing an Emulab machine for WAN experiments. This work was supported in part by a Google Faculty Research Award and Buffalo Blue Sky Initiative. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding sources.

References

[1] GMP – The GNU multiple precision arithmetic library. `http://www.gmplib.org`.

[2] M. Abspoel, A. Dalskov, D Escudero, and A. Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. IACR Cryptology ePrint Archive Report 2019/1298, 2019.

[3] M. Abspoel, D. Escudero, and N. Volgushev. Secure training of decision trees with continuous attributes. In *PoPETS*, pages 167–187, 2021.

[4] N. Agrawal, A. Shahin Shamsabadi, M. J. Kusner, and A. Gascón. QUOTIENT: Two-party secure neural network training and prediction. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1231–1247, 2019.

[5] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM Conference on Computer and Communications Security (CCS)*, pages 805–817, 2016.

[6] D. Beaver and A. Wool. Quorum-based secure multi-party computation. In *EUROCRYPT*, pages 375–390, 1998.

[7] G. R. Blakley. Safeguarding cryptographic keys. In *International Workshop on Managing Requirements Knowledge (MARK)*, pages 313–318, 1979.

[8] M. Blanton, A. Kang, and C. Yuan. Improved building blocks for secure multi-party computation based on secret sharing with honest majority. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 377–397, 2020.

[9] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security (ESORICS)*, pages 192–206, 2008.

[10] O. Catrina and S. De Hoogh. Improved primitives for secure multiparty integer computation. In *International Conference on Security and Cryptography for Networks (SCN)*, pages 182–199, 2010.

[11] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. ASTRA: High throughput 3PC over rings with application to secure prediction. In *ACM Workshop on Cloud Computing Security (CCSW)*, pages 81–92, 2019.

[12] H. Chaudhari, R. Rachuri, and A. Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *NDSS*, 2020.

[13] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO*, pages 769–798, 2018.

[14] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography Conference (TCC)*, pages 342–362, 2005.

[15] A. Dalskov, D. Escudero, and M. Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. IACR Cryptology ePrint Archive Report 2020/1330, 2020.

[16] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *IEEE Symposium on Security and Privacy*, pages 1102–1120, 2019.

[17] I. Damgård, C. Orlandi, and M. Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In *CRYPTO*, pages 799–829, 2018.

[18] I. Damgard, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

[19] I. Damgård and R. Thorbek. Efficient conversion of secret-shared values between different fields. IACR Cryptology ePrint Archive Report 2008/221, 2008.

[20] H. Eerikson, M. Keller, C. Orlandi, P. Pullonen, J. Puura, and M. Simkin. Use your brain! arithmetic 3PC for any modulus with active security. In *Conference on Information-Theoretic Cryptography (ITC)*, pages 5:1–5:24, 2020.

[21] R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111, 1998.

[22] J. Geusebroek, G. J. Burghouts, and A. Smeulders. The Amsterdam Library of Object Images. *International Journal of Computer Vision*, 61(1):103–112, 2005.

[23] M. Ito, A. Saito, and T. Nishizeki. Secret sharing schemes realizing general access structures. In *IEEE Global Telecommunication Conference (Globecom)*, pages 99–102, 1987.

[24] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security Symposium*, pages 1651–1669, 2018.

[25] M. Keller. MP-SPDZ: A versatile framework for multi-party computation. In *ACM Conference on Computer and Communications Security (CCS)*, page 15751590, 2020.

[26] L. Kerik, P. Laud, and J. Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In *International Conference on Financial Cryptography and Data Security Workshops*, pages 271–287, 2016.

[27] Y. LeCun and C. Cortes. MNIST handwritten digit database. `http://yann.lecun.com/exdb/mnist/`, 2010.

[28] Y. Lindell and A. Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest majority. In *ACM Conference on Computer and Communications Security (CCS)*, pages 259–276, 2017.

[29] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *ACM Conference on Computer and Communications Security (CCS)*, pages 619–631, 2017.

[30] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. IACR Cryptology ePrint Archive Report 2017/452, 2017.

[31] U. Maurer. Secure multi-party computation made simple. In *Security in Communication Networks (SCN)*, pages 14–28, 2002.

[32] S. Micali, O. Goldreich, and A Wigderson. How to play any mental game. In *ACM Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.

[33] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. DELPHI: A cryptographic inference service for neural networks. In *USENIX Security Symposium*, pages 2505–2522, 2020.

[34] P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. In *ACM Conference on Computer and Communications Security (CCS)*, pages 35–52, 2018.

[35] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE Symposium on Security and Privacy*, pages 19–38, 2017.

[36] A. Patra and A. Suresh. BLAZE: Blazing fast privacy-preserving machine learning. In *NDSS*, 2020.

[37] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. CrypTFlow2: Practical 2-party secure inference. In *ACM Conference on Computer and Communications Security (CCS)*, pages 325–342, 2020.

[38] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Asia Conference on Computer and Communications Security (ASIACCS)*, pages 707–721, 2018.

[39] SecureSCM. Deliverable D9.2, EU FP7 Project Secure Supply Chain Management (SecureSCM). `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.221.393&rep=rep1&type=pdf`, 2009.

[40] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[41] S. Wagh, D. Gupta, and N. Chandran. SecureNN: Efficient and private neural network training. *Proceedings of Privacy Enhancing Technologies Symposium (PoPETS)*, 2019(3):26–49, 2019.

[42] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–270, 2002.

[43] Y. Zhang, A. Steele, and M. Blanton. PICCO: A general-purpose compiler for private distributed computation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 813–826, 2013.

A Additional Protocols

Protocol 6 shows the computation associated with evaluating the MNIST neural network model from [29]. Protocol 7 provides computation associated with support vector machine evaluation on q classes and m features. For the ALOI SVM from [16], we use $q = 463$ and $m = 128$.

Protocol 6 MNIST neural network evaluation

1: (Convolution) 28×28 input image, 5×5 window size, $(1,1)$ stride, 16 output channels: $\mathbb{Z}_{2^{20}}^{16\times 576} \leftarrow \mathsf{MatMult}(\mathbb{Z}_{2^{20}}^{16\times 25}, \mathbb{Z}_{2^{20}}^{25\times 576})$;
2: (ReLU) calculates ReLU for each entry of $\mathbb{Z}_{2^{20}}^{16\times 576}$;
3: (Max Pooling) input $\mathbb{Z}_{2^{20}}^{16\times 576}$ and $1 \times 2 \times 2$ window size, outputs $\mathbb{Z}_{2^{20}}^{16\times 12\times 12}$;
4: (Conversion) $\mathbb{Z}_{2^{30}}^{16\times 144} \leftarrow \mathsf{Convert}(\mathbb{Z}_{2^{20}}^{16\times 144}, 20, 30)$;
5: (Convolution) 5×5 window size, $(1,1)$ stride, 16 output channels: $\mathbb{Z}_{2^{30}}^{16\times 64} \leftarrow \mathsf{MatMult}(\mathbb{Z}_{2^{30}}^{16\times 400}, \mathbb{Z}_{2^{30}}^{400\times 64})$;
6: (ReLU) calculates ReLU for each entry of $\mathbb{Z}_{2^{30}}^{16\times 64}$;
7: (Max Pooling) input $\mathbb{Z}_{2^{30}}^{16\times 64}$ and $1 \times 2 \times 2$ window size, outputs $\mathbb{Z}_{2^{30}}^{16\times 4\times 4}$;
8: (Conversion) $\mathbb{Z}_{2^{40}}^{16\times 16} \leftarrow \mathsf{Convert}(\mathbb{Z}_{2^{30}}^{16\times 16}, 30, 40)$;
9: (Fully Connected) Connects 256 incoming nodes to 100 outgoing nodes : $\mathbb{Z}_{2^{40}}^{100\times 1} \leftarrow \mathsf{MatMult}(\mathbb{Z}_{2^{40}}^{100\times 256}, \mathbb{Z}_{2^{40}}^{256\times 1})$;
10: (ReLU) calculates ReLU for each entry of $\mathbb{Z}_{2^{40}}^{100\times 1}$;
11: (Conversion) $\mathbb{Z}_{2^{49}}^{100\times 1} \leftarrow \mathsf{Convert}(\mathbb{Z}_{2^{40}}^{100\times 1}, 40, 49)$;
12: (Fully Connected) Connects 100 incoming nodes to 10 outgoing nodes : $\mathbb{Z}_{2^{49}}^{10\times 1} \leftarrow \mathsf{MatMult}(\mathbb{Z}_{2^{49}}^{10\times 100}, \mathbb{Z}_{2^{49}}^{100\times 1})$;
13: **return** $\mathbb{Z}_{2^{49}}^{10\times 1}$;

Protocol 7 SVM classification, where q is the number of classes, m is the number of features, $f_{i,j}$ is the feature vector, and b_j are the biases

1: **for each** $j = 1$ to q **in parallel do** $[c_j] \leftarrow [b_j] + \sum_{i=1}^{m} \mathsf{DotProd}([f_{i,j}], [x_i])$;
2: **return** $([a_{\text{ind}}], [\text{ind}]) \leftarrow \mathsf{ArgMax}([c_1], \ldots, [c_q])$;