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

Alessandro Baccarini
University at Buffalo (SUNY)
Buffalo, New York, USA
anbaccar@buffalo.edu

Marina Blanton
University at Buffalo (SUNY)
Buffalo, New York, USA
mblanton@buffalo.edu

Chen Yuan
University at Buffalo (SUNY)
Buffalo, New York, USA
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 $\mathbb{Z}_{2^k}$ can boost performance. In this work, we develop a suite of multi-party protocols 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 that our techniques are substantially faster than their field-based equivalents when instantiated with a different number of parties and perform on par with or better than state-of-the-art techniques with designs customized for a fixed number of parties. We evaluate our techniques on machine learning applications and show that they offer attractive performance.

KEYWORDS

secure multi-party computation, replicated secret sharing, privacy-preserving machine learning

1 INTRODUCTION

Secure multi-party computation has recently seen notable performance improvements 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 highlighted secret sharing techniques which were often previously overlooked in the literature. Secret sharing (SS) offers superior performance for arithmetic operations such as matrix multiplications over other cryptographic tools, and has been extensively used for privacy-preserving neural network (NN) inference and training [14, 15, 18, 27, 36, 47, 49, 55, 56]. Because SS offers information-theoretic security, computation can proceed on short integers, aiding efficiency.

Traditionally, performance of SS 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. Traditional techniques such as Shamir SS [54] 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 $\mathbb{Z}_{2^k}$ [8, 12, 16, 20]. Unfortunately, Shamir SS – 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 alternatives.

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 relative to stronger settings, 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., [8, 14, 15, 41, 47]) and cannot tolerate collusion. This means that the techniques do not easily generalize to a larger number of participants, should there be a need to change the computation setup, e.g., to permit the use of a higher collusion threshold. This is the task we set to address in this work and generalize computation based on replicated secret sharing (RSS) to support more than $n = 3$ computational parties.

Our contributions can be summarized as follows:

- We design a comprehensive set of elementary building blocks for RSS 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, multiplication followed by reconstruction as a single building block, denoted by MulPub, and inputting private values into computation. 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 custom results which apply to the three-party setting only [8, 55].
- We build on the techniques of [20] and [27] to develop higher-level protocols over $\mathbb{Z}_{2^k}$ such as 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. We observe that when $n = 3$ our ring-based techniques can be between 10 and 33 times faster than their field-based counterparts for different types of operations. Incorporating recent advances in random bit generation can yield even more promising results. The improvement from switching to ring-based techniques decreases as

Proceedings on Privacy Enhancing Technologies 2023(1), 608–626
© 2023 Copyright held by the owner/author(s).
https://doi.org/10.56553/popets-2023-0035

the number of parties n grows, but with $n = 7$ we can still observe runtime improvements by a factor of 2 or higher for certain operations.
- We improve the techniques of [18] for securely evaluating quantized NNs and eliminate the need for fixed-point multiplication and large truncation, which enables us to use a significantly smaller ring.
- We also evaluate performance of our techniques on machine learning applications, namely, NN predictions and quantized NN inference. Similarly, our runtimes are significantly faster than similar field-based implementations and compare favorably to the state of the art designed to work with a fixed number of parties.

For RSS-based techniques, it is expected that they will be used with a relatively small n. This is similar to most efficient techniques based on Shamir SS (e.g., [11, 13]) which also rely on RSS for certain operations.

2 RELATED WORK

Secret sharing [10, 54] is a popular choice for secure multi-party computation, and common options include Shamir SS [54], additive SS, and RSS [31] for three parties. Computation over rings, and specifically $\mathbb{Z}_{2^k}$, has recently gained attention in publications including [5, 8, 12, 16, 18, 20, 22, 26, 34, 41]. We can distinguish between three-party techniques based on RSS such as [5, 8, 12, 22, 26, 34, 41]; multi-party techniques based on additive SS such as [16, 20], often 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 [33] and others.

The first category is the closest to this work and includes Sharemind [12], a well-developed framework for three-party computation with a single corruption using custom protocols; Araki et al. [8] who use three-party with a single corruption to support arithmetic or Boolean circuits; and several compilers from passively secure to actively secure protocols [5, 22, 26, 41]. Dalskov et al. [19] also studied 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 RSS over a ring. While RSS 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 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. [20] for some of our protocols. While this work is for the SPD$\mathbb{Z}_{2^k}$ framework [16] in the malicious setting with no honest majority, once we develop elementary building blocks, the structure of higher-level protocols can remain similar. Composite protocols such as comparison, conversion, and truncation require a large number of random bits. We leverage the edaBit protocol from [27] to efficiently generate sets of binary and arithmetic shared bits. Their technique improves upon the daBit technique [52]. Rabbit [44] builds on daBits [52] and edaBits [27] and developed an efficient n-party comparison protocol by relying on commutativity of addition over fields and rings. Their protocol offers significant improvement over [27] in most adversarial settings over a field, but remains comparable with a passively secure honest majority over a ring.

Framework	Setting		Techniques			No. of	Networks	
	S-H	Mal	HE	GC	SS	Parties	[42]	qMob.
SecureML [48]	✓		✓	✓	✓	2	✓	
MiniONN [42]	✓		✓	✓	✓	2	✓	
Gazelle [33]	✓		✓	✓	✓	2	✓	
DELPHI [46]	✓		✓	✓	✓	2	✓	
Chameleon [51]	✓			✓	✓	2	✓	
CrypTFflow [50]	✓	✓			✓	2	✓	
CrypTFflow2 [38]	✓		✓		✓	3		
SecureNN [55]	✓				✓	3	✓	
Falcon [56]	✓	✓			✓	3	✓	
ASTRA [14]	✓			✓	✓	3		
BLAZE [49]	✓	✓		✓	✓	3		
ABY3 [47]	✓	✓		✓	✓	3		
SecureQ8 [18]	✓	✓			✓	3, n^*	✓	✓
Trident [15]	✓	✓		✓	✓	4		
Fantastic Four [19]	✓	✓			✓	4		
This Work	✓				✓	n	✓	✓

Table 1: Comparison of state-of-the-art PPML frameworks. (*) [18] supports n parties in the semi-honest, honest majority setting over a field $\mathbb{F}_p$, but only three parties over a ring. The two NNs we consider are [42]'s four-layer convolutional NN, and the quantized version of MobileNets (qMob.) [30].

Literature on PPML is also related to this work, and we present a high-level comparison of the current state-of-the-art in Table 1. Each framework is subdivided according to their security assumptions (semi-honest or malicious), the cryptographic techniques used, the number of parties supported, and the methods of evaluation. We highlight several key works below.

We distinguish between two-party solutions, where one party holds the model and the other 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 [42] and Gazelle [33], both of which studied NN evaluation using SS, homomorphic encryption (HE), and garbled circuits (GC).

Multi-party constructions provide protocols for training and inference across multiple parties. ABY3 [47] combines techniques based on replicated and binary SS with GCs in the three-party setting with honest majority. SecureNN [55] provides three-party protocols for a variety of NN functions under the same security assumption as ABY3. Their protocols are asymmetric, where parties have dedicated roles in a computation. This work is improved upon with FALCON [56] by adding malicious security with honest majority and combining the techniques from SecureNN and ABY3.

ASTRA [14] is a three-party framework that uses SS over the ring $\mathbb{Z}_{2^k}$ under both semi-honest and malicious security assumptions. Similar to SecureNN, protocols are asymmetric. Abspoel et al. [6] apply the MP-SPDZ [34] framework for secure outsourced training of decision trees. Their system operates under the three-party, honest-majority assumption with RSS. Dalskov et al. [18] were the first to address quantized NN inference using secure multi-party computation. Their system is built into MP-SPDZ and benchmarked on the MobileNets [30] network architecture. Keller et al. [36] conducts quantization-based training and inference with three parties and one semi-honest corruption.

3 PRELIMINARIES

3.1 Secure Multi-Party Computation

We consider a 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 semi-honest participants and use simulation-based security (see Appendix B for detail).

As customary with SS techniques, the set of computational parties does not have to coincide with (and can be formed independently of) 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 reveal no information to that party. Consequently, if we wish to design a functionality that takes secret-shared input and produces shares of the output, any computational party should learn nothing from protocol execution.

3.2 Secret Sharing

A SS 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 SS 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 are linear SS schemes, which have the property that a linear combination of secret shared values can be performed locally on the shares. Examples of linear SS schemes include additive SS with $x = \sum_i x_i$ (as used in Sharemind [12] with $n = 3$ and in SPDZ [23] with any n), Shamir SS which realizes (n, t) secret sharing with $t < n/2$ and represents a share as evaluation of a polynomial on a distinct point, and RSS, which we discuss next.

3.3 Replicated Secret Sharing

Our techniques utilize RSS [31] 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 sets 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 consequently describe in Section 4 the way we realize these operations. 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 information-theoretic) setting as proposed in [17]; see Section 4 for detail.

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. In addition to adding secret-shared values, we 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. Addition $[a] + b$ converts b to $[b]$ without using randomness (e.g., we could set one share to b and the remaining shares to 0 to maintain $\sum_{T \in \mathcal{T}} b_T = b$). Multiplication $[c] = [a] \cdot b$ sets $c_T = a_T \cdot b$ (in $\mathcal{R}$) $\forall T \in \mathcal{T}$.

For convenience and without loss of generality, we let $n = 2t + 1$. 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.

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 [17].

$\mathsf{PRG}(s_T)$ 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 $\mathsf{PRF} : \mathcal{R} \times \{0, 1\}^\kappa \rightarrow \mathcal{R}$ 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 G.next to retrieve the next (secret-shared) element from G. Similarly, if $\mathsf{G}_T = \mathsf{PRG}(s_T)$, notation G_T.next refers to the next pseudo-random share output by G_T.

PRandR. $[a] \leftarrow$ PRandR() computes a secret-shared random element of ring $\mathcal{R}$. We implement this function by executing PRG($[k]$).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 $[c] \leftarrow$ Mul($[a], [b]$) (or simply $[a]{\cdot}[b]$) is realized using the fact that $[a]{\cdot}[b] = \sum_{T_1,T_2 \in \mathcal{T}} a_{T_1}{\cdot}b_{T_2}$ (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 $\rho : \mathcal{T}\times\mathcal{T} \longrightarrow [1, n]$ 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 across 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 [9, 45] 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 $p \notin T$). 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, e.g., [8] 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 as well. However, if this idea is applied naively, it results in unnecessarily high overhead. In particular, if we instruct each party p to generate all shares for its secret, some shares will be known to more than t participants and thus do not contribute to secrecy. Instead, our solution eliminates shares that p does not possess and thus do not contribute to secrecy. Thus, our construction utilizes key material consistent with the setup of the RSS scheme. In particular, we use the same key setup as in pseudorandom secret sharing, 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, such that this information is public knowledge.

In addition to the mapping ρ, our multiplication protocol requires another mapping $\chi : [1, n] \rightarrow \mathcal{T}$, which specifies for each party p 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 $\rho$ and $\chi$
1:  each $p \in [1, n]$ does the following
2:    let $S_p = \{T \in \mathcal{T} \mid p \notin T\}$;
3:    $v^{(p)} = \sum_{T_1,T_2 \in \mathcal{T}, \rho(T_1,T_2)=p} a_{T_1} b_{T_2}$;
4:    $v^{(p)}_{\chi(p)} = v^{(p)}$;
5:    for $T \in S_p$ do $c_T = 0$;
6:    for $p' \in [1, n]$ in order do
7:      for $T \in S_p$ do
8:        if $(p' \neq p) \wedge (p' \notin T) \wedge (\chi(p') \neq T)$ then
9:          $c_T = c_T + \mathrm{G}_T$.next;
10:       else if $(p' = p) \wedge (\chi(p) \neq T)$ then
11:         $z = \mathrm{G}_T$.next;
12:         $c_T = c_T + z$;
13:         $v^{(p)}_{\chi(p)} = v^{(p)}_{\chi(p)} - z$;
14:       end if
15:     end for
16:   end for
17:   send $v^{(p)}_{\chi(p)}$ to each $p' \notin \chi(p)$ (other than itself);
18:   for $p' \in [1, n]$ such that $p \notin \chi(p')$ do
19:     receive $v^{(p')}_{\chi(p')}$ from $p'$, set $c_{\chi(p')} = c_{\chi(p')} + v^{(p')}_{\chi(p')}$;
20:   end for
21:   $c_{\chi(p)} = c_{\chi(p)} + v^{(p)}_{\chi(p)}$;
22: 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 sends t ring elements and draws $\binom{n-1}{t} - 1 + (n-1)\binom{n-2}{t} - t$ pseudo-random ring elements (which is $(t+1)(\binom{n-1}{t} - 1)$ when $n = 2t+1$). 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 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.

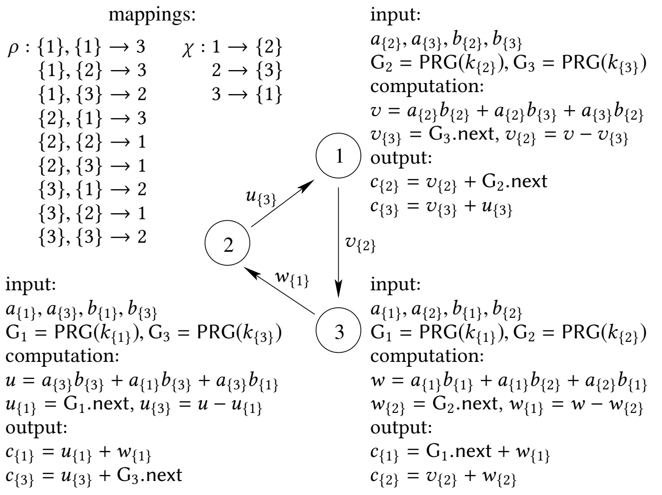

mappings:
$\rho : \{1\},\{1\} \to 3$, $\{1\},\{2\} \to 3$, $\{1\},\{3\} \to 2$, $\{2\},\{1\} \to 3$, $\{2\},\{2\} \to 1$, $\{2\},\{3\} \to 1$, $\{3\},\{1\} \to 2$, $\{3\},\{2\} \to 1$, $\{3\},\{3\} \to 2$
$\chi : 1 \to \{2\}$, $2 \to \{3\}$, $3 \to \{1\}$
input: $a_{\{2\}}, a_{\{3\}}, b_{\{2\}}, b_{\{3\}}$
$G_2 = \mathsf{PRG}(k_{\{2\}}), G_3 = \mathsf{PRG}(k_{\{3\}})$
computation: $v = a_{\{2\}}b_{\{2\}} + a_{\{2\}}b_{\{3\}} + a_{\{3\}}b_{\{2\}}$
$v_{\{3\}} = G_3.\mathsf{next}$, $v_{\{2\}} = v - v_{\{3\}}$
output: $c_{\{2\}} = v_{\{2\}} + G_2.\mathsf{next}$
$c_{\{3\}} = v_{\{3\}} + u_{\{3\}}$
1
2
3
$u_{\{3\}}$
$v_{\{2\}}$
$w_{\{1\}}$
input: $a_{\{1\}}, a_{\{3\}}, b_{\{1\}}, b_{\{3\}}$
$G_1 = \mathsf{PRG}(k_{\{1\}}), G_3 = \mathsf{PRG}(k_{\{3\}})$
computation: $u = a_{\{3\}}b_{\{3\}} + a_{\{1\}}b_{\{3\}} + a_{\{3\}}b_{\{1\}}$
$u_{\{1\}} = G_1.\mathsf{next}$, $u_{\{3\}} = u - u_{\{1\}}$
output: $c_{\{1\}} = u_{\{1\}} + w_{\{1\}}$
$c_{\{3\}} = u_{\{3\}} + G_3.\mathsf{next}$
input: $a_{\{1\}}, a_{\{2\}}, b_{\{1\}}, b_{\{2\}}$
$G_1 = \mathsf{PRG}(k_{\{1\}}), G_2 = \mathsf{PRG}(k_{\{2\}})$
computation: $w = a_{\{1\}}b_{\{1\}} + a_{\{1\}}b_{\{2\}} + a_{\{2\}}b_{\{1\}}$
$w_{\{2\}} = G_2.\mathsf{next}$, $w_{\{1\}} = w - w_{\{2\}}$
output: $c_{\{1\}} = G_1.\mathsf{next} + w_{\{1\}}$
$c_{\{2\}} = v_{\{2\}} + w_{\{2\}}$

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

Compared to other results, the three-party version of our protocol matches communication of recent multiplication from [8], which is available only for three parties and improves on communication of Sharemind's three-party multiplication from [37] by a factor of 2. For multi-party multiplication it can be desirable to use a different communication pattern when a designated party reconstructs a protected value and communicates it to others (as in, e.g., [21]) which scales better as n grows. However, our version has lower communication when $n = 3$, uses fewer rounds, and n is typically small with RSS.

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 $\{2\}$) 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, with its proof available in Appendix B:

Theorem 1. *Multiplication $[c] \leftarrow [a]\cdot[b]$ is secure according to definition 1 in the (n, t) setting with $n = 2t+1$ in the presence of secure communication channels and assuming* PRG *is a pseudo-random generator.*

Our multiplication protocol shares conceptual similarities with (optimized) multiplication from [35]. In particular, both sample pseudorandom secret shares according to the access structure and communicate a single (properly protected) element to a number of other participants. Our solution explicitly defines all maps and the computation associated with computing each share of the output, while the latter appears to be under-specified in [35].

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 multi-variate 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 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.

Table 2 shows performance of these and other basic protocols for the general (n, t) and the (3,1) settings. Communication is measured as the number of ring elements sent by each party and computation is the number of cryptographic operations (i.e., retrieval of the next pseudo-random element using a PRG) per party.

4.3 Revealing Private Values

Open. Reconstruction of a secret shared value $a = \mathsf{Open}([a])$ amounts to communicating missing shares to each party such 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, each party receives $d = \binom{n}{t} - \binom{n-1}{t}$ missing shares during this operation.

Our next observation is that when n is not small (such as when $n = 7$), the value of d will exceed n and transmitting d messages to each party is not needed. Since 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 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 SS 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.

MulPub. Functionality $c = \mathsf{MulPub}([a], [b])$ refers to multiplying two secret-shared $[a]$ and $[b]$ and opening their product c. We

Operation	Rounds	(3, 1) setting		(n, t) setting	
		Comm	Crypto ops	Comm	Crypto ops
PRG([s]).next, PRandR()	0	0	2	0	$\binom{n-1}{t}$
Mul([a], [b])	1	1	2	t	$(t+1)\left(\binom{n-1}{t}-1\right)$
Open([a])	1	1	0	t	0
MulPub([a], [b])	1	2	2	$n-1$	$\binom{n-1}{t}$
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 2: Performance of basic RSS operations with computation and communication per party.

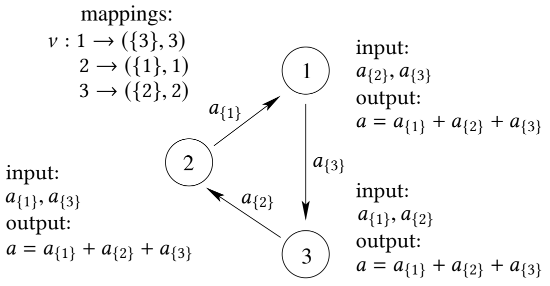

mappings:
$v: 1 \to (\{3\}, 3)$
$2 \to (\{1\}, 1)$
$3 \to (\{2\}, 2)$
input: $a_{\{2\}}, a_{\{3\}}$ output: $a = a_{\{1\}} + a_{\{2\}} + a_{\{3\}}$
input: $a_{\{1\}}, a_{\{3\}}$ output: $a = a_{\{1\}} + a_{\{2\}} + a_{\{3\}}$
input: $a_{\{1\}}, a_{\{2\}}$ output: $a = a_{\{1\}} + a_{\{2\}} + a_{\{3\}}$
$a_{\{1\}}$
$a_{\{2\}}$
$a_{\{3\}}$
1
2
3

Figure 2: Sample three-party Open([a])**; arithmetic is in** $\mathcal{R}$**.**

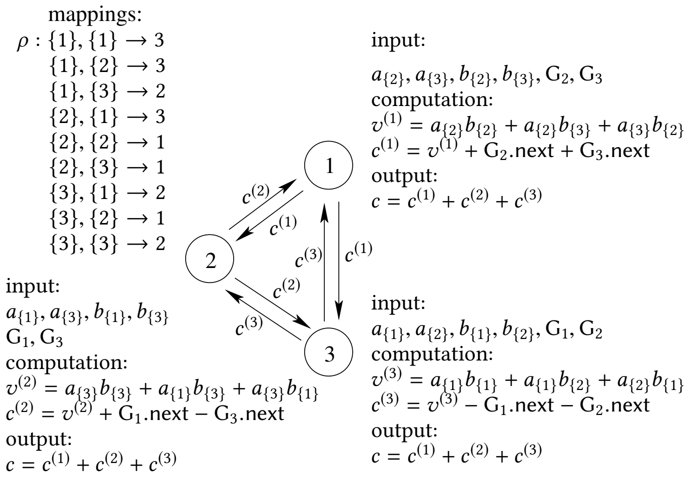

mappings:
$\rho: \{1\},\{1\} \to 3$
$\{1\},\{2\} \to 3$
$\{1\},\{3\} \to 2$
$\{2\},\{1\} \to 3$
$\{2\},\{2\} \to 1$
$\{2\},\{3\} \to 1$
$\{3\},\{1\} \to 2$
$\{3\},\{2\} \to 1$
$\{3\},\{3\} \to 2$
input: $a_{\{2\}}, a_{\{3\}}, b_{\{2\}}, b_{\{3\}}, G_2, G_3$
computation: $v^{(1)} = a_{\{2\}}b_{\{2\}} + a_{\{2\}}b_{\{3\}} + a_{\{3\}}b_{\{2\}}$
$c^{(1)} = v^{(1)} + G_2.\text{next} + G_3.\text{next}$
output: $c = c^{(1)} + c^{(2)} + c^{(3)}$
input: $a_{\{1\}}, a_{\{3\}}, b_{\{1\}}, b_{\{3\}}$ G_1, G_3
computation: $v^{(2)} = a_{\{3\}}b_{\{3\}} + a_{\{1\}}b_{\{3\}} + a_{\{3\}}b_{\{1\}}$
$c^{(2)} = v^{(2)} + G_1.\text{next} - G_3.\text{next}$
output: $c = c^{(1)} + c^{(2)} + c^{(3)}$
input: $a_{\{1\}}, a_{\{2\}}, b_{\{1\}}, b_{\{2\}}, G_1, G_2$
computation: $v^{(3)} = a_{\{1\}}b_{\{1\}} + a_{\{1\}}b_{\{2\}} + a_{\{2\}}b_{\{1\}}$
$c^{(3)} = v^{(3)} - G_1.\text{next} - G_2.\text{next}$
output: $c = c^{(1)} + c^{(2)} + c^{(3)}$
$c^{(2)}$
$c^{(1)}$
$c^{(3)}$
$c^{(1)}$
$c^{(2)}$
$c^{(3)}$
1
2
3

Figure 3: Sample three-party MulPub([a], [b])**; arithmetic is in** $\mathcal{R}$**.**

discuss this functionality because in the past, this operation could be implemented more efficiently than multiplication followed by an opening in alternative SS frameworks (e.g., see [13]), 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 calculated value is no longer random and must be re-randomized prior to opening it. In our RSS setting, this is realized by relying on parties locally computing pseudo-random values. Specifically, we associate a secret key k_T with each $T \in \mathcal{T}$ (i.e., this is the same key shares used with PRandR() and multiplication) and use pseudo-random values G_T.next to protect the share of the product that each party locally computes, prior to that party revealing its randomized value to all others. We require all blinding pseudo-random values sum to 0 to ensure the reconstructed product is correct. In the three-party case, this can be achieved by adding some pseudo-random values and subtracting others, as illustrated in 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 protocol, each party draws the same number of elements from each G_T in its possession to ensure that after a single protocol execution all parties are in the same state (but a party may discard some 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

Protocol 2 $c \leftarrow$ MulPub([a], [b])

```
// pre-distributed values are $[k]$ and public map $\rho$
1:  each $p \in [1, n]$ does the following:
2:      $v^{(p)} = c^{(p)} = \sum_{T_1,T_2 \in \mathcal{T}, \rho(T_1,T_2)=p} a_{T_1} b_{T_2}$;
3:      for $T \in S_p$ do
4:          let $j$ be the number of parties $p' < p$ for $p' \notin T$;
5:          for $i = 0$ to $t - 1$ do
6:              $z = G_T$.next;
7:              if $j = t$ then $c^{(p)} = c^{(p)} - z$;
8:              else if $i = j$ then $c^{(p)} = c^{(p)} + z$;
9:              end if
10:         end for
11:     end for
12:     send $c^{(p)}$ to all other parties, and set $c = c^{(p)}$;
13:     for $i = 1$ to $n - 1$ do
14:         receive $c^{(p')}$ from distinct $p'$, set $c = c + c^{(p')}$;
15:     end for
16: return $c$;
```

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* $\mathsf{MulPub}([a], [b])$ *is secure according to definition 1 in the* (n, t) *setting with* $n = 2t + 1$ *assuming* PRG *is a pseudo-random generator.*

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 G_T.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 (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 $v^{(1)} + z_{\{3\}}$, $v^{(2)} - z_{\{3\}}$, 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. The full proof is given in Appendix B.

Similar to multiplication, MulPub can be generalized to evaluate any (multi-variate) polynomial of degree 2 and open the result.

4.4 Inputting Private Values

There will be a need to enter private values into the computation in subsequent protocols, and we defer two variants of this functionality – when input is provided by an external party and one of the computational parties – to Appendix A.

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 $\mathbb{Z}_{2^k}$ is the primary reason for supporting secure computation over rings because it enables utilization of native CPU instructions for ring operations.

The goal of this work is to enable efficient general-purpose computation over rings $\mathbb{Z}_{2^k}$, we therefore 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 power of 2). Combined, 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.

Because a number of protocols for common operations over $\mathbb{Z}_{2^k}$ have already been developed, some of the constructions that we mention in these sections are adaptations of prior protocols to our setting and we defer their specification to the appendix. In particular, Appendix A provides specification of random bit generation protocol, RandBit, that produces a bit shared in $\mathbb{Z}_{2^k}$ and a more recent version from [27], edaBit, that generates a number (k in our case) of random bits r_i shared in $\mathbb{Z}_2$ together with a representation of the bits as an integer $r = \sum_{i=1}^{k} 2^i r_i$ shared in $\mathbb{Z}_{2^k}$. The former can be computed in a single round, while the latter uses noticeably lower communication per bit, but the round complexity is logarithmic in k and t.

We also describe a comparison algorithm for computing $[a] \leq [b]$, which is commonly implemented by determining the most significant bit of the difference between a and b and denoted by MSB. Performance of these protocols is summarized in Tables 3 and 13.

Truncation is a necessary building block when working with fixed-point values or simulating fixed-point computation using integer arithmetic and permits us to minimize the ring size. Starting from [13], probabilistic truncation of input a by m bits that produces $\lfloor a/2^m \rfloor + u$, where u is a bit, is significantly faster than precise truncation that rounds down. It is biased towards rounding to the nearest integer to $a/2^m$ and is sufficient for our purpose. The protocol we present, $\mathsf{TruncPr}([a], m)$, is a constant-round solution that combines the approach from [18] with edaBits from [27] and inherits from [27] the requirement that input a is 1 bit shorter than the ring size, i.e., $\mathsf{MSB}(a) = 0$. We use notation $[x]_\ell$ to denote that SS is over $\mathbb{Z}_{2^\ell}$.

The truncation protocol, given as Protocol 3, uses related random values r and $\hat{r}$, bit decomposition of which are known, where $r = \sum_{i=0}^{k-1} 2^i r_i$ is a full-size random value and $\hat{r} = \sum_{i=m}^{k-1} 2^i r_i$ is the portion remaining after truncating m bits. We thus modify the edaBit protocol to produce those values simultaneously. Each $[r]$ and $[\hat{r}]$ is computed as a sum of $t + 1$ integers, so we must compensate for two types of carries: (i) addition of m least significant bits in r will produce carry bits into the next bits which are not accounted for in $\hat{r}$ and (ii) while the carry bits past the k bits are automatically removed in the ring when computing r, these bits remain in $\hat{r}$ due to its shorter length. Because we compute the bitwise representation of r using bitwise addition protocol BitAdd, we can also extract the carry bit into any desired position which is already computed during the addition. The logic of the truncation protocol necessitates the removal of the $(k-1)$th bit. For this reason, we capture carries into the mth and $(k-1)$th positions and denote those bits from the ith call to BitAdd as $\mathrm{cr}_{i,m}$ and $\mathrm{cr}_{i,k-1}$, respectively (line 10). We subsequently convert the $2\log(t+1)$ carry bits and the most significant bit of r, denoted as b_{k-1}, from shares over $\mathbb{Z}_2$ to $\mathbb{Z}_{2^k}$ using binary-to-arithmetic sharing protocol B2A (from [20]). All interactive operations except the last one (line 20) can be precomputed. Security follows from the protocol logic as specified in prior work and from security of the building blocks.

It is also possible to use the above protocol to truncate an input $[a]$ by a private number of bits $[m]$ as outlined in [18]: Let M be some public upper bound on m. Protocol $\mathsf{TruncPriv}([a], [m], M)$ then needs to securely compute $[2^{M-m}] \cdot [a]$ and can subsequently call $\mathsf{TruncPr}([2^{M-m} \cdot a], M)$. A performance summary is given in Table 3.

Protocol	Rand. Protocol	Rounds	Communication
$\mathsf{MSB}([a]_k)$	RandBit	$\log(k-1)+3$	$2t(k+3)$
	edaBit	$\log(t+1)(\log(k)+1)+\log(k-1)+4$	$t^2(\log(k)+1)+7t+1/2$
$\mathsf{TruncPr}([a]_k, m)$	RandBit	2	$t(2k+1)$
	edaBit	$\log(t+1)(\log(k)+1)+4$	$t^2(\log(k)+2/k+4)+t(1/k+4)+1/2$
$\mathsf{Convert}([a]_k, k, k')$	RandBit	$\log(k)+4$	$2t(k+k')+t(\log(k)+2)$
	edaBit	$\log(t+1)(\log(k)+1)+\log(k)+3$	$t^2(\log(k)+1)+t(2k'+\log(k)+3)$

Table 3: Performance of composite protocols with communication measured in the number of ring elements sent per party over $\mathbb{Z}_{2^{k+2}}$ for RandBit and $\mathbb{Z}_{2^k}$ for edaBit(k).

```
Protocol 3 $[a/2^m]_k \leftarrow \mathsf{TruncPr}([a]_k, m)$
 1: for $p = 1, \ldots, t+1$ in parallel do
 2:     party $p$ samples $r_0^{(p)}, \ldots, r_{k-1}^{(p)} \in \mathbb{Z}_2$ and computes $r^{(p)} = \sum_{j=0}^{k-1} r_j^{(p)} 2^j$ and $\hat{r}^{(p)} = \sum_{j=m}^{k-1} r_j^{(p)} 2^j$;
 3:     simultaneously execute $[r^{(p)}]_k \leftarrow \mathsf{Input}(r^{(p)}, k)$, $[\hat{r}^{(p)}]_k \leftarrow \mathsf{Input}(\hat{r}^{(p)}, k)$, and $[r_i^{(p)}]_1 \leftarrow \mathsf{Input}(r_i^{(p)}, 1)$ for $i$=1, . . ., $k$, with $p$ being the input owner;
 4: end for
 5: $[r]_k = \sum_{p=1}^{t+1} [r^{(p)}]_k$; $[\hat{r}]_k = \sum_{p=1}^{t+1} [\hat{r}^{(p)}]_k$;
 6: $s = t+1$;
 7: for $i = 1, \ldots, \lceil \log(t+1) \rceil$ do
 8:     for $j = 1, \ldots, \lfloor s/2 \rfloor$ in parallel do
 9:         $\ell = j + s \cdot (i-1)$;
10:         $\langle [r_1^{(j)}]_1, \ldots, [r_{k-1}^{(j)}]_1 \rangle, [\mathrm{cr}_{\ell,m-1}]_1, [\mathrm{cr}_{\ell,k}]_1 \leftarrow \mathsf{BitAdd}(\langle [r_1^{(2j-1)}]_1, \ldots, [r_{k-1}^{(2j-1)}]_1 \rangle, \langle [r_1^{(2j)}]_1, \ldots, [r_{k-1}^{(2j)}]_1 \rangle)$;
11:         if $s$ mod 2 = 0 then $s = s/2$;
            else
12:             $\langle [r_1^{(\frac{s+1}{2})}]_1, \ldots, [r_{k-1}^{(\frac{s+1}{2})}]_1 \rangle = \langle [r_1^{(s)}]_1, \ldots, [r_{k-1}^{(s)}]_1 \rangle$;
13:             $s = (s+1)/2$;
14:         end if
15:     end for
16: end for
17: $[b_0]_1, \ldots, [b_{k-1}]_1 = [r_0^{(1)}]_1, \ldots, [r_{k-1}^{(1)}]_1$;
18: $[b_{k-1}]_k, \langle [\mathrm{cr}_{\ell,m}]_k, \rangle, \langle [\mathrm{cr}_{\ell,k-1}]_k \rangle \leftarrow \mathsf{B2A}([b_{k-1}]_1, \langle [\mathrm{cr}_{\ell,m}]_1 \rangle, \langle [\mathrm{cr}_{\ell,k-1}]_1 \rangle)$ for $\ell$=1, . . . , $t$;
19: $[\hat{r}]_k = [\hat{r}]_k - [b_{k-1}]_k \cdot 2^{k-m-1} + \sum_{\ell=1}^{t} ([\mathrm{cr}_{\ell,m}]_k - [\mathrm{cr}_{\ell,k-1}]_k 2^{k-m-1})$;
20: $c \leftarrow \mathsf{Open}([a]_k + [r]_k)$;
21: $c' = (c/2^m) \bmod 2^{k-m-1}$;
22: $[b]_k = (c/2^{k-1}) + [b_{k-1}]_k - 2(c/2^{k-1})[b_{k-1}]_k$;
23: return $c' - [\hat{r}]_k + [b]_k \cdot 2^{k-m-1}$;
```

6 NEURAL NETWORK APPLICATIONS

Today it is typical to benchmark secure multi-party frameworks on machine learning applications, e.g., NN inference. We briefly introduce NN basics and describe two mechanisms for improving efficiency of secure NN inference.

A *neural network* is a series of interconnected layers consisting of neurons. Each neuron has an associated weight and bias used for computation on some input data and outputs a prediction based on that data. A NN network layer takes the form $\mathbf{y} = g(\mathbf{x}\mathbf{W} + \mathbf{b})$, where $\mathbf{x}$ is the input vector from the previous layer, $\mathbf{W}$ is the weight tensor, $\mathbf{b}$ is the bias vector, and g is some activation function. Sample activation functions are Rectified Linear Unit (ReLU), which on input $\mathbf{x} = (x_1, \ldots, x_N)$ computes $\mathbf{y} = (y_1, \ldots, y_N)$ where each $y_i = \max(0, x_i)$, and its variant ReLU6 which computes $y_i = \min(\max(0, x_i), 6)$.

```
Protocol 4 $[a]_{k'} \leftarrow \mathsf{Convert}([a]_k, k, k')$, where $k' > k$
1: $[r]_k, [r_0]_1, \ldots, [r_{k-1}]_1 \leftarrow \mathsf{edaBit}(k)$;
2: $c \leftarrow \mathsf{Open}([a]_k - [r]_k)$;
3: $[a_0]_1, \ldots, [a_{k-1}]_1 \leftarrow \mathsf{BitAdd}(c, [r_0]_1, \ldots, [r_{k-1}]_1)$;
4: for $i = 0$ to $k-1$ in parallel $[a_i]_{k'} \leftarrow \mathsf{B2A}([a_i]_1, k')$;
5: return $[a]_{k'} = \sum_{i=0}^{k-1} [a_i]_{k'} 2^i$;
```

6.1 Share Conversion

Conventional NN evaluation uses floating-point arithmetic, while secure evaluations for performance reasons typically employ fixed-point computation or emulate it on integers. If inputs are represented in the form of fixed-length integers, the values will grow with each layer that performs matrix multiplication. This can impact on performance because comparison-based activation and pooling operations have cost linear in the bitlength of ring elements. For this reason, it can be advantageous to start with a smaller ring size and increase it mid-computation to accommodate longer values.

This approach involves converting secret-shared $[a]_k$ over $\mathbb{Z}_{2^k}$ to a different representation $[a]_{k'}$ over $\mathbb{Z}_{2^{k'}}$ for $k' > k$. Conversion techniques between certain types of fields are known [24], but they do not apply to our case. Simply casting k-bit shares to k'-bit shares for $k' > k$ affects correctness because the overflow due to share addition is not reduced modulo 2^k. Thus, the task is to leave k least significant bits of the value and erase the remaining bits in a longer share representation. One way to achieve this is to invoke truncation as $([a] \cdot 2^{k'-k}) \gg 2^{k'-k}$ or $[a] - ([a] \gg k)2^k$. However, because computing precise truncation is costlier for rings than fields, 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 Convert and given as Protocol 4 using edaBits. An equivalent version can be constructed using RandBit. It is based on bit decomposition from [20] and uses Boolean to arithmetic conversion, B2A, from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{k'}}$ and bitwise integer addition, BitAdd. Performance is summarized in Table 3.

6.2 Quantized Neural Networks

To improve efficiency of NN inference, it is common to employ quantization, which makes the resulting models suitable for deployment in constrained environments and is a well-studied field (see, e.g., [29]). We outline the standard quantization approach from [32] and its privacy-preserving realization from [18] for quantized TFLite models and consequently describe our optimizations.

For a vector $\mathbf{x}$, each real-valued x_i is represented as $x_i = m(\bar{x}_i - z)$, where $m \in \mathbb{R}$ is the scale and z and $\bar{x}_i$ are 8-bit integers with z being the zero point. Given an input column vector $\mathbf{x} = (x_1, \ldots, x_N)$ and a row vector $\mathbf{w} = (w_1, \ldots, w_N)$ of $\mathbf{W}$ with quantization parameters (m_1, z_1) and (m_2, z_2), respectively, the dot product of $\mathbf{x}$ and $\mathbf{w}$, $y = \sum_{i=1}^{N} x_i w_i$, is specified with quantization parameters (m_3, z_3). Since $y \approx m_3 \cdot (\bar{y} - z_3)$, $x_i \approx m_1 \cdot (\bar{x}_i - z_1)$, and $w_i \approx m_2 \cdot (\bar{w}_i - z_2)$, quantized $\bar{y}$ is computed as $\bar{y} \approx z_3 + m_1 m_2 / m_3 \cdot \sum_{i=1}^{N} (\bar{x}_i + z_1) \cdot (\bar{w}_i - z_2) = z_3 + m \cdot s$. Computing s requires integer-only arithmetic and is guaranteed to fit in $16 + \log N$ bits. The scale $m = m_1 m_2 / m_3$ is a small real number. It can be written as $m = 2^{-e} m'$ with normalized $m' \in [0.5, 1)$ which informs the value of e and represented as a 32-bit integer m'', where $m' \approx 2^{-31} m''$.

Two-dimensional convolutions typically add a quantized bias $\bar{b}$ once the dot product is computed. This is handled by setting the scale of the bias to $m_1 m_2$ and the zero-point to 0, such that the bias can be added to s prior to scaling. The last step of a convolution layer is to apply an activation function such as ReLU6. In a quantized NN, this functions as a clamping operation which eliminates values outside of range $[0, 255]$ and uses $m_3 = 6/255$ and $z_3 = 0$. This guarantees correct range while maximizing precision with 8-bit quantized values. Going forward, m_3 becomes m_1 for the next layer and thus all intermediate layers share the same $m_1 = m_3 = 6/255$. Other activation functions such as sigmoid would be handled differently, but we only consider clamping-based functions like [18].

Computing the convolution layer securely requires the model owner to enter private quantization parameters into the computation, including all zero points z_i, modified scale m'', and integer scale adjustment 2^{M-e-31}, where M is an upper bound set to 63. After privately computing the dot product $[s]$ and adding the bias vector $[\bar{b}]$, the result is multiplied by $[m'']$ and need to be truncated by private amount $31 + e$. The truncation is accomplished by multiplying the scaled dot product by $[2^{M-n-31}]$ and $[m \cdot s]$ and consequently truncating by M bits. Lastly, after adding $[z_3]$ locally, clamping the result to the interval $[0, 255]$ is performed using two comparisons.

A limitation of [18]'s approach is it required large scaling factors and consequently a large ring size of $k = 72$ for working with real numbers, using M-bit truncation with $M = 63$. We propose a modified approach where scales are folded into other aspects of the layer computation and conduct smaller truncation at the end of each layer, which guarantees compact representation of intermediate results.

Let superscript $\langle i \rangle$ denote the layer number. Starting from layer 0, the entire layer computation (dot product, scaling, and clamping) can be interpreted as computing $0 \leq \bar{y}^{\langle 0 \rangle} \leq 255$, where

$$\bar{y}^{\langle 0 \rangle} = \frac{m_1^{\langle 0 \rangle} m_2^{\langle 0 \rangle}}{m_3^{\langle 0 \rangle}} \left(\left(\sum_{i=1}^{N} (\bar{x}_i^{\langle 0 \rangle} - z_1^{\langle 0 \rangle}) \cdot (\bar{w}_i^{\langle 0 \rangle} - z_2^{\langle 0 \rangle}) \right) + \bar{b}^{\langle 0 \rangle} \right)$$

and $z_3^{\langle i \rangle}$ was set to 0, as prescribed by the clamping operation, for all layers except the last one. Because $m_3^{\langle 0 \rangle} = 6/255$, we scale the equation to redefine $\bar{y}^{\langle 0 \rangle}$ as

$$\bar{y}^{\langle 0 \rangle} = \sum_{i=1}^{N} \left(\bar{x}_i^{\langle 0 \rangle} - z_1^{\langle 0 \rangle} \right) \cdot \left(\bar{w}_i^{\langle 0 \rangle} - z_2^{\langle 0 \rangle} \right) + \bar{b}^{\langle 0 \rangle},$$

where $0 \leq \bar{y}^{\langle 0 \rangle} \leq 6/m_1^{\langle 0 \rangle} m_2^{\langle 0 \rangle}$. Now, our clamping operation can use these bounds, with the upper bound being privately entered by the model owner to avoid division. As before, the output of this layer becomes the input for the subsequent layer, i.e., $\bar{x}^{\langle i \rangle} = \bar{y}^{\langle i-1 \rangle}$. Our modified incoming vector, denoted $\hat{x}^{\langle 1 \rangle}$, is coupled with an additional scaling factor of $(255 m_1^{\langle 0 \rangle} m_2^{\langle 0 \rangle})/6$, such that $\bar{x}^{\langle 1 \rangle} = 255 m_1^{\langle 0 \rangle} m_2^{\langle 0 \rangle} \hat{x}^{\langle 1 \rangle}/6 = \delta^{\langle 1 \rangle} \hat{x}^{\langle 1 \rangle}$. Using $\bar{x}^{\langle 1 \rangle} = \delta^{\langle 1 \rangle} \hat{x}^{\langle 1 \rangle}$ gives us

$$\bar{y}^{\langle 1 \rangle} = \left(\sum_{i=1}^{N} (\hat{x}_i^{\langle 1 \rangle} - z_1^{\langle 1 \rangle}/\delta^{\langle 1 \rangle}) \cdot (\bar{w}_i^{\langle 1 \rangle} - z_2^{\langle 1 \rangle}) \right) + \bar{b}^{\langle 1 \rangle}/\delta^{\langle 1 \rangle}$$

with $0 \leq \bar{y}^{\langle 1 \rangle} \leq 6/(\delta^{\langle 1 \rangle} m_1^{\langle 1 \rangle} m_2^{\langle 1 \rangle})$. This expression can be evaluated securely without needing fixed-point multiplication or large truncation, and all bounds are computed by the model owner prior to privately entering them in the computation.

Evaluating subsequent layers in this fashion causes the outputs to grow by factor $\delta^{\langle i+1 \rangle} = \delta^{\langle i \rangle} 255 m_1^{\langle i \rangle} m_2^{\langle i \rangle}/6$ with $\delta^{\langle 0 \rangle} = 1$. However, we can ensure values remain small by truncating the output $\bar{y}^{\langle i+1 \rangle}$ by $\ell^{\langle i \rangle}$ bits. With the right choice of $\ell^{\langle i \rangle}$ we are able to maintain the necessary accuracy, and the value of $\delta^{\langle i+1 \rangle}$ consequently becomes $\delta^{\langle i+1 \rangle} = \delta^{\langle i \rangle} \cdot 255 m_1^{\langle i \rangle} m_2^{\langle i \rangle}/(6 \cdot 2^{\ell^{\langle i \rangle}})$. The maximum number of bits we can truncate in a layer needs to comply with constraint $\delta^{\langle i \rangle} \cdot 255 m_1^{\langle i \rangle} m_2^{\langle i \rangle}/(6 \cdot 2^{\ell^{\langle i \rangle}}) \geq 1$, which leads to $\ell^{\langle i \rangle} \leq \left\lfloor \log_2 (255 \delta^{\langle i \rangle} m_1^{\langle i \rangle} m_2^{\langle i \rangle}/6) \right\rfloor$. Once again, these values are independent of the input data and become a part of the model. We thus can use TruncPriv outlined in Section 5 for truncation by a private amount. The net result is that we are able to use a significantly smaller bound M and consequently substantially shorter ring size k. In practice, the coefficients introduced in our methodology can reasonably be folded into the scaling factors m themselves.

Other layers such as average pooling can be approximated by substituting the division by some integer d with truncation by $\lfloor \log d \rfloor$ bits, and softmax can be replaced with argmax when computing the final prediction. These changes can slightly impact the scaling factors, but have no impact on the accuracy since we leverage basic algebraic properties, without changing the fundamental calculation itself.

7 PERFORMANCE EVALUATION

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

	Setup		Batch Size							Comm.
			1	10	10^2	10^3	10^4	10^5	10^6	
3PC	$\mathbb{FG}$	30	0.081	0.0893	0.245	1.47	12.1	120	1,236	4
		60	0.082	0.0912	0.255	1.61	12.9	127	1,289	8
	$\mathcal{R}$	30	0.075	0.079	0.097	0.153	0.606	5.87	59.6	4
		60	0.075	0.076	0.108	0.320	1.096	9.68	113	8
5PC	$\mathbb{FG}$	30	0.124	0.158	0.384	2.37	16.8	159	1,550	8
		60	0.129	0.167	0.439	2.45	17.9	173	1,669	16
	$\mathbb{FD}$	30	0.224	0.267	0.836	3.74	34.1	23.5	2,227	6.4*
		60	0.229	0.278	0.924	4.01	36.4	25.4	2,436	12.8*
	$\mathcal{R}$	30	0.184	0.170	0.278	0.711	4.00	41.3	377	8
		60	0.221	0.224	0.326	0.943	6.36	63.9	579	16
7PC	$\mathbb{FG}$	30	0.168	0.198	0.497	3.17	24.5	238	2,353	12
		60	0.174	0.224	0.541	3.47	27.7	257	2,520	24
	$\mathbb{FD}$	30	0.275	0.327	1.18	7.69	60.4	502	4,829	6.9*
		60	0.281	0.354	1.34	8.01	67.8	534	5,186	13.7*
	$\mathcal{R}$	30	0.254	0.37	0.72	2.84	25.2	266	2,536	12
		60	0.285	0.40	0.81	4.13	34.9	365	3,490	24

Table 4: Runtime of multiplication protocols in ms and communication is per party per operation in bytes (* means average for asymmetric communication patterns). $\mathbb{FG}$ and $\mathbb{FD}$ refer to the optimized GRR and DN field multiplication from [11], resp., and $\mathcal{R}$ is our ring realization. 30 and 60 are integer bitlengths.

	Setup	Matrix Size			
		10×10	100×100	500×500	1000×1000
3PC	$\mathbb{F}$ (30)	0.318	91.6	1,025	8,289
	$\mathbb{F}$ (60)	0.319	94.2	1,187	8,723
	$\mathcal{R}$ (30)	0.187	2.83	212	1567
	$\mathcal{R}$ (60)	0.288	3.82	226	1638
5PC	$\mathbb{FG}$ (30)	0.457	95.2	1,145	8,927
	$\mathbb{FG}$ (60)	0.462	97.9	1,321	10,134
	$\mathbb{FD}$ (30)	1.07	97.4	1,273	9,995
	$\mathbb{FD}$ (60)	1.09	102	1,493	11,964
	$\mathcal{R}$ (30)	0.253	11.7	720	5,224
	$\mathcal{R}$ (60)	0.331	12.5	813	5,939
7PC	$\mathbb{FG}$ (30)	0.891	97.7	1,272	9,953
	$\mathbb{FG}$ (60)	0.904	101	1,478	10,864
	$\mathbb{FD}$ (30)	1.29	99.8	1,483	11,569
	$\mathbb{FD}$ (60)	1.35	104	1,536	13,742
	$\mathcal{R}$ (30)	0.658	48.0	5,880	48,793
	$\mathcal{R}$ (60)	0.705	59.0	7,509	71,234

Table 5: Runtime of matrix multiplication in ms.

The implementation was done in C++ and is available at [4]. 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 computation parties.

All experiments use identical 2.4 GHz virtual machines with 26 GB of RAM. They were connected via 10 Gbps Ethernet links, which we throttled to 1 Gbps using the `tc` command. Two-way latency was measured to be 0.106 ms. All experiments use a single core. WAN benchmarks can be found in Appendix D.

7.1 Micro-benchmarks

In this section we report performance of individual operations such as multiplication, matrix multiplication, random bit generation (RandBit and edaBit) and comparison (MSB). The experiments used two bitlengths, $k = 30$ and $k = 60$, which allows us to use the `uint32_t` and `uint64_t` integer types, respectively, to implement ring operations.

Tables 4 and 5 report performance of multiplication and matrix multiplication, respectively. As we strive to measure performance improvement when we switch computation from a field to a ring, we compare performance of our protocols to those using Shamir SS in the same setting (i.e., semi-honest security with honest majority) using PICCO implementation [57] with recent improvements to multiplication from [11]. The field size is set to accommodate 30- and 60-bit integers. Batch size denotes how many operations were executed at the same time in a single batch.

We measure runtime and communication with a number of parties ranging from 3 to 7. For field multiplication, we measure performance of two variants: GRR-based with higher asymptotic communication and 1 round ($\mathbb{FG}$) and DN-based with lower asymptotic communication and 2 rounds ($\mathbb{FD}$) as described in [11]. The former is strictly better in the three-party setting. The latter, despite its lower communication, does not lead to better performance as the number of parties increases as it internally relies on RSS. However, the difference in performance of the two variants is not substantial enough to play a major role in larger computations, as is demonstrated in Table 5. We therefore proceed with $\mathbb{FG}$ with 3 parties and $\mathbb{FD}$ with 5–7 parties in other experiments where multiplication is used.

From Table 4 we observe that our RSS performance is up to 20 times faster with a sufficiently large batch size in the 3-party setting compared to the field and some performance advantage is maintained even with 7 parties despite the need to compute with a much larger number of shares. Note that the performance gain is due to faster instructions because communication is comparable across different variants. This indicates that using native CPU instructions for secure arithmetic has remarkable advantage.

Matrix multiplication in Table 5 is performed in a single round using the necessary number of dot-products. Because local work is the bottleneck, we see performance improvement by up to a factor of 32.3 after switching to a ring with 3 parties. Performance improvement with 5 parties is by up to a factor of 8.3 and up to a factor of 2.1 with 7 parties. The ring performance is superior for all configurations evaluated except for the two largest matrices with 7 parties.

Tables 6 and 7 provide random bit generation results. To support k-bit integers, ring-based RandBit requires ring $\mathbb{Z}_{2^{k+2}}$. Field-based RandBit from [13] does not increase the field size; however, all uses of RandBit we are aware of are for operations such as comparisons that utilize statistical hiding and, as a result, increase the field size by a statistical security parameter κ (typically set to 48 in implementations). For this reason, our field-based RandBit and MSB

	Setup	Batch Size							Comm.
		1	10	10^2	10^3	10^4	10^5	10^6	
3PC	$\mathbb{F}$ (30)	0.104	0.158	0.457	2.87	25.4	259	2,637	20
	$\mathbb{F}$ (60)	0.107	0.164	0.546	3.47	32.8	336	3,480	28
	$\mathcal{R}$ (30)	0.124	0.111	0.156	0.330	2.37	21.8	249	8
	$\mathcal{R}$ (60)	0.112	0.124	0.170	0.555	4.57	43.9	477	16
5PC	$\mathbb{F}$ (30)	0.175	0.281	0.815	5.97	50.8	506	4,985	40
	$\mathbb{F}$ (60)	0.171	0.291	0.869	6.75	65.4	66.1	6,794	56
	$\mathcal{R}$ (30)	0.169	0.178	0.234	0.595	4.50	45.9	468	16
	$\mathcal{R}$ (60)	0.262	0.244	0.356	1.252	8.39	88.1	854	32
7PC	$\mathbb{F}$ (30)	0.249	0.369	1.15	8.14	70.6	684	6,842	60
	$\mathbb{F}$ (60)	0.264	0.412	1.34	9.42	84.9	824	8,251	84
	$\mathcal{R}$ (30)	0.255	0.268	0.472	1.53	10.4	117	1,134	24
	$\mathcal{R}$ (60)	0.237	0.288	0.508	2.15	18.3	217	2,092	48

Table 6: Runtime of RandBit **protocols in ms and communication is per party per operation in bytes.**

	Protocol	Batch Size							Comm.
		1	10	10^2	10^3	10^4	10^5	10^6	
3PC	$\mathcal{R}$ (30)	0.564	0.577	0.832	2.96	20.5	207	1,978	32
	$\mathcal{R}$ (60)	0.622	0.737	1.111	5.66	43.2	405	4,175	68
	[3] (32)	19.7	15.9	16.2	16.7	20.0	138	1,368	–
	[3] (64)	22.8	25.5	25.2	24.4	30.6	254	2,201	–

Table 7: Runtime of edaBit **protocols in ms compared to MP-SPDZ implementation. Communication for our solution is per party per operation in bytes.**

	Protocol	Batch Size							Comm.
		1	10	10^2	10^3	10^4	10^5	10^6	
3PC	$\mathbb{F}$ (30)	1.29	3.71	23.7	206	2,051	21.8s	222s	624
	$\mathbb{F}$ (60)	1.97	7.51	54.7	471	4,654	46.7s	487s	864
	$\mathcal{R}$+rB (30)	0.71	0.74	1.54	9.23	88.7	0.85s	8.25s	265
	$\mathcal{R}$+rB (60)	0.76	1.01	3.92	29.2	322	3.04s	30.0s	1009
	$\mathcal{R}$+eB (30)	1.23	1.24	1.51	4.14	30.7	0.27s	2.77s	57
	$\mathcal{R}$+eB (60)	1.31	1.46	1.88	7.88	60.6	0.56s	5.71s	117
	[3]+eB (32)	23.3	23.1	22.9	23.5	27.3	0.18s	1.31s	–
	[3]+eB (64)	34.2	31.6	33.4	32.5	35.9	0.25s	2.15s	–
	[3]+ABY3 (32)	8.51	8.97	9.05	13.6	52.1	0.39s	3.66s	–
	[3]+ABY3 (64)	9.09	9.06	8.88	14.2	58.5	0.41s	3.87s	–
5PC	$\mathbb{F}$ (30)	2.12	6.17	37.5	349	3,219	32.2s	333s	1248
	$\mathbb{F}$ (60)	3.32	11.9	84.0	738	7,021	68.8s	701s	1728
	$\mathcal{R}$+rB (30)	1.62	1.83	3.87	21.0	197	1.82s	18.6s	530
	$\mathcal{R}$+rB (60)	2.09	2.59	8.08	70.7	644	6.10s	60.5s	2018
	$\mathcal{R}$+eB (30)	3.77	4.02	6.33	27.1	203	1.97s	19.1s	162
	$\mathcal{R}$+eB (60)	4.16	4.71	10.1	56.3	447	4.24s	41.1s	338
7PC	$\mathbb{F}$ (30)	3.08	9.14	48.4	452	4.42s	43.2s	447s	1872
	$\mathbb{F}$ (60)	4.55	13.1	101	943	9.36s	94.2s	959s	2592
	$\mathcal{R}$+rB (30)	2.08	2.74	7.38	56.1	0.62s	5.95s	65.4s	795
	$\mathcal{R}$+rB (60)	2.39	3.93	18.6	190	1.75s	17.6s	179s	3027
	$\mathcal{R}$+eB (30)	5.52	7.64	25.8	186	1.65s	16.8s	165s	316
	$\mathcal{R}$+eB (60)	6.31	10.6	45.3	371	3.57s	36.3s	356s	663

Table 8: Runtime of MSB **protocols in ms unless marked otherwise. Communication is per party per operation in bytes. rB and eB indicate variants using RandBit and edaBit, respectively.**

benchmarks utilize 79- and 109-bit fields. Both versions of RandBit in Table 6 communicate the same number of field or ring elements; however, the performance gain of the ring version grows as we increase the batch size, reaching 10 to 12-fold improvement with 3 and 5 parties and indicating that local field-based computation is the bottleneck. This is in large part due to the need to perform modulo exponentiations (see [13]). That is, even though field-based RandBit also relies on RSS, other non-RSS computation such as modulo exponentiation is significant and the overall slowdown with the number of parties is not as large. In the 7-party setting the improvement of the ring-based variant is by up to a factor of 6.

The concept of edaBit is recent and for that reason in Table 7 we compare our implementation to that reported in the original publication [27], available through MP-SPDZ repository [3]. Note that each edaBit corresponds to generating k random bits together with the corresponding k-bit random integer. It is clear from the table that MP-SPDZ's implementation is optimized for large sizes and fast networks. In particular, it gives comparable runtime for batches of size 1 and 1,000. For the same reason, we were unable to accurately report communication cost per operation from the experiments and refer the reader to the original publication [27] for that information. Note that the times we measured for MP-SPDZ are very different from those originally provided in [27], which reported the ability to generate 7.18 million 64-bit edaBits per second. This is over 15 times faster than the fastest time per operation we record and stems from the differences in hardware. In particular, experiments in [27] were run multi-threaded on powerful AWS `c5.9xlarge` instances with 36 cores and a 10 Gbps link. This distinction highlights the need to reproduce experiments on similar hardware to draw meaningful comparisons about performance of different algorithms.

Table 8 reports performance of multiple MSB protocols: (i) field-based protocol from [13] using PICCO's implementation with optimizations from [11], our ring implementations (ii) using RandBit and (iii) using edaBit, and ring-based implementations from MP-SPDZ [3] (iv) using edaBit and (v) using ABY3. The last two support only three-party computation.

The gap between the first two shows performance improvement due to switching from field-based to ring-based arithmetic. Both of them make a linear in k number of calls to RandBit, but our implementation executes BitLT over $\mathbb{Z}_2$, while field-based uses a fixed field for all operations. As a result, our ring RandBit-based MSB is up to 26.9 times faster than the field version with 3 parties, up to 17.9 times with 5 parties, and up to 7.2 times with 7 parties.

If we compare our RandBit and edaBit MSB implementations, the use of the edaBit version becomes advantageous starting from batch sizes of 100 with 3 parties, 1000–10000 with 5 parties, but is not beneficial with 7 parties. This can be explained by the need to perform a larger number of bitwise additions during edaBit generation as the number of computational parties increases.

MP-SPDZ's edaBit-based implementation in the three-party setting generally took longer to run than our edaBit-based implementation until the batch size becomes large. As explained earlier, this is due to different performance emphases in the two implementations.

ABY3 (three-party) implementation is slower than what we obtain except for the largest batch sizes with the longer bitlengths.

We also visualize time per operation with variable batch sizes in Figure 4 using three parties. Multiplication and RandBit sub-figures compare ring vs. field protocols, indicating a substantial gap as expected; edaBit sub-figure compares our and MP-SPDZ implementations in the same setting; and MSB sub-figure compares RandBit and edaBit variants.

It is also informative to compare our field vs. ring results with those of SPDZ. While SPDZ [23] and its ring version SPD$\mathbb{Z}_{2^k}$ [16, 20] use a much stronger adversarial model and different type of SS, we would like to know whether similar savings are achievable in different settings. [20] reports that performance improved by a factor of 4.6–4.9 for multiplication and by a factor of 5.2–6.0 for RandBit-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 20 times for multiplication and up to 26.9 improvement for MSB. This may be explained by the fact that our techniques are more lightweight and perhaps switching to faster arithmetic makes less of an impact in the SPDZ setting.

7.2 Machine Learning Applications

We next evaluate our protocols on machine learning applications and show that they exhibit good performance. We consider NNs and quantized NNs, in part to facilitate comparison to prior work.

Neural Networks. There are many types of NNs, and for our standard benchmarking we chose the NN from MiniONN [42] for the MNIST dataset [39] (Figure 12 in [42], Network B in [55], and Network C in [56]), because it is a popular choice for evaluating privacy-preserving NN inference. The MNIST NN evaluation uses convolution, fully-connected layers, an ReLU activation function, and max pooling of a window 2×2 to compute the maximum element in that window.

We use MiniONN's implementation choices and, in particular, run the computation on integer inputs. To avoid using floating-point arithmetic, [42] scaled inputs by a factor of 1000 and rounded to the nearest integer. To compensate for the bitlength of the intermediate results growing with each multiplication, [42]'s implementation ran the computation using a 37-bit modulus and avoided the use of truncation. However, we determined that this size is too small, and 49 bits are needed to correctly evaluate the model, which we subsequently use. Our implementation achieves the same 99.0% precision as reported for this model in [43] (which corrects [42]).

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 use significantly shorter integers than 49 bits. Because the cost of comparisons is linear in the bitlength of the ring elements, we can substantially improve performance by starting computation on shorter values and converting the intermediate results to a larger ring prior to multiplication, which increases the size of the intermediate results. Therefore, we start computation with 20-bit integers and increase the ring size by 10 bits prior to subsequent matrix multiplications.

Performance of MNIST NN inference with three parties (total time) is presented in Table 9. We also ran the same computation over a field (using [11, 57]), which required an 89-bit modulus. To closely mimic our ring-based implementation, this implementation computes with integers of increasing sizes, but uses the same modulus throughout the computation.

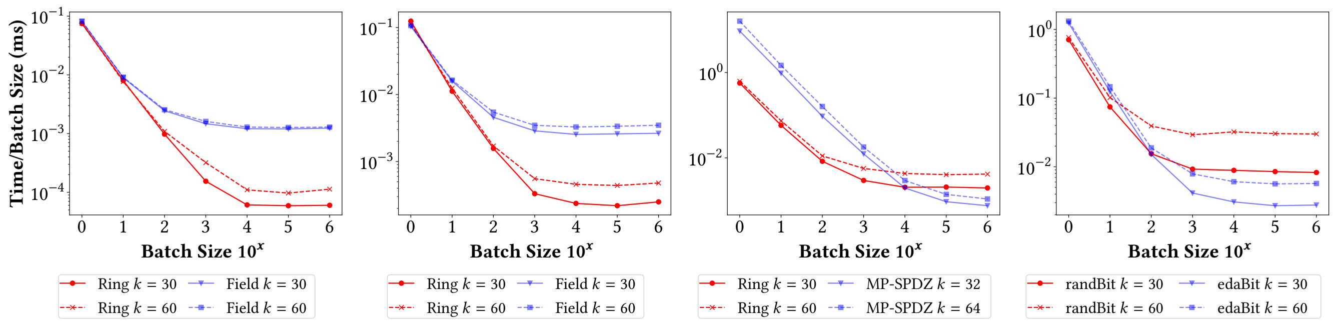
We also include runtimes of two-party MiniONN [42], two-party Gazelle [33], two-party FALCON [40], SecureNN with custom three-party arithmetic [55], three-party FALCON [56], and three-party Dalskov et al. [18] with two types of truncation (TruncPr and TruncPrSp, respectively). Many of those solutions were executed on more powerful hardware which would not lead to a meaningful performance comparison. For that reason, we reproduced the implementations except for MiniONN, Gazelle, and two-party FALCON [40] on our machines. From those, only Gazelle was executed on more powerful AWS instances with multi-threading at the time of original publication, but its performance even with that setup is not competitive with what we achieve. Furthermore, the solution was consequently surpassed in SecureNN, which we execute on our hardware.

Table 9 shows the time for a single inference and for executing multiple inferences in a batch where available. We can see that our single prediction time is lower than in other publications despite the fact that the solution is generalizable to a larger number of parties with a larger collusion threshold. Our communication is also low and the only construction that improves the time when executing multiple predictions in parallel is FALCON [56]. While their implementation benefits from larger batching through multi-threading and lower communication due to small moduli, FALCON is limited to three parties. Our solution, however, can be invoked with a larger number of parties as demonstrated in Table 10 with $n = 5$.

Several other publications benchmarked NN predictions [7, 14, 15, 38, 46, 47, 49–51]. However, because they do not support or do not run MiniONN's MNIST NN evaluation, we cannot directly compare our performance. For example, while ABY^3 [47] is said to use MiniONN's MNIST NN, evaluation is actually based on a different, simpler model used in Chameleon [51].

Quantized Neural Networks. Benchmarks for quantized NNs were based on the MobileNets [30] architecture, which consists of 28 layers and 1000 output classes. The network alternates between 3×3 depthwise convolutions and 1×1 pointwise convolutions. A resolution multiplier ρ (128–224) scales the dimensions of the input image, and a width multiplier α (0.25–1.0) scales the size of the input and output channels. The models we used are hosted on TensorFlow's online repository [2] and are trained on the ImageNet [25] dataset. We experimentally determined that an upper bound of $M = 16$ is sufficient for truncation by a private value, since all computed $\ell^{\langle i \rangle}$s are ≤ 9 for all model configurations.

Performance of quantized MobileNets inference is presented in Tables 11 and 12 with 3 and 5 parties, respectively. Our methodology from Section 6.2 allowed us to reduce the ring size from $k = 72$ to $k = 30$ or less, potentially reducing the time by a factor of 2. For accurate comparison, we executed [18]'s implementation on our machines using the same setting. Since a 5-party honest-majority ring implementation is not available in [18], or more generally in MP-SPDZ, we use a field-based implementation for the 5-party case from MP-SPDZ. Recall that the ability to generalize ring-based



(a) Mult($[a]$, $[b]$) **(b)** RandBit() **(c)** edaBit(k) **(d)** MSB($[a]$)

Figure 4: Three-party micro-benchmarks results.

	Field	MiniONN*	Gazelle*	SecureNN	FALCON* [40]	FALCON [56]		[18]	Ours, 3PC			
Batch Size	1	1	1	1	1	1	128	1	1	5	10	50
Latency	1328	9320	810	1228	840	123	20.4	279, 196	82.5	68.8	67.4	67.6
Comm.	8.12	657.5	70.0	37.9	92.5	0.55		15.6, 9.7	2.76			

Table 9: Runtime of MNIST NN prediction in ms and communication in MB. (*) denotes results taken from the original publications.

	Field, 5PC	Ours, 5PC			
Batch Size	1	1	5	10	25
Latency (ms)	2047	414	370	367	355
Comm. (MB)	16.2*	6.34			

Table 10: Performance of MNIST NN prediction in 5-party configuration. (*) means average for asymmetric communication.

		Ours				MP-SPDZ $\mathbb{Z}_{2^k}$, [18]			
	α	0.25	0.5	0.75	1.0	0.25	0.5	0.75	1.0
ρ	128	3.19	6.47	9.92	13.3	3.19	6.26	9.88	14.0
	160	4.94	10.0	15.1	20.7	4.15	8.17	13.6	19.3
	192	7.17	14.3	22.0	29.7	5.00	11.0	17.8	26.7
	224	9.71	19.9	30.0	40.9	6.57	14.1	23.1	34.9

Table 11: Performance of 3PC quantized MobileNets prediction in seconds. MP-SPDZ results are over a ring $\mathbb{Z}_{2^k}$.

		Ours				MP-SPDZ $\mathbb{F}_p$, [18]			
	α	0.25	0.5	0.75	1.0	0.25	0.5	0.75	1.0
ρ	128	23.7	48.0	73.1	98	442	688	992	1343
	160	37.4	75.1	113	151	904	1414	2031	2765
	192	52.8	107	162	219	1398	2182	3156	4269
	224	72.7	145	220	297	1919	3005	4324	5877

Table 12: Performance of 5PC quantized MobileNets prediction in seconds. MP-SPDZ results are over a field $\mathbb{F}_p$.

honest-majority protocols to more participants is our main objective.

The results our 3-party solution achieves are comparable to those in [18] despite ring reduction and can be explained by the differences in the algorithms. That is, Escudero et al. [28] experimentally determined that [18]'s implementation with ABY3's local conversion was superior to edaBits (which we use) only in one setting that we use (semi-honest, honest majority setting over $\mathbb{Z}_{2^k}$). In addition, MP-SPDZ's optimization for large computation also aids its efficiency. This demonstrates that our quantized NN solution can aid efficiency. Furthermore, our gain in the 5-party case is significant, leading to the reduction in time by a factor of 13–26.

8 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 field-based computation. We design low-level operations for n-party replicated secret sharing over any ring and consequentially build on them to enable general-purpose protocols over ring $\mathbb{Z}_{2^k}$. Our implementation results demonstrate that ring-based implementations of different operations are significantly faster than their field-based equivalents with 3, 5, and even 7 parties. This allows us to improve performance of different applications including privacy-preserving machine learning tasks. We specifically test performance of neural network and quantized neural network classification and determine 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 [42]. The authors also would like to thank Marcel Keller for help with MP-SPDZ experiments and several constructions included in MP-SPDZ. This work was supported in part by a Google Faculty Research Award, Buffalo Blue Sky Initiative, and US National Science Foundation grant 2213057. 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] OpenSSL – Cryptography and SSL/TLS toolkit. https://www.openssl.org/. Version: 1.1.1.

[2] Tensorflow repository. https://tensorflow.org/lite/guide/hosted_models. Last accessed: 6/14/22.

[3] MP-SPDZ repository. https://github.com/data61/MP-SPDZ, 2021. Commit: 5ab8c702dde2f25ae7f2f2d0e4d47f5d716fa621.

[4] Replicated secret sharing over a ring. https://github.com/anbaccar/RSS_ring_ppml, 2022. Commit: d921581401301c35660e15aaf329f41436699389.

[5] M. Abspoel, A. Dalskov, D. Escudero, and A. Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 122–152, 2021.

[6] M. Abspoel, D. Escudero, and N. Volgushev. Secure training of decision trees with continuous attributes. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2021(1):167–187, 2021.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] H. Chaudhari, R. Rachuri, and A. Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[16] 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 *Advances in Cryptology – CRYPTO*, pages 769–798, 2018.

[17] 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.

[18] A. Dalskov, D. Escudero, and M. Keller. Secure evaluation of quantized neural networks. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2020(4):355–375, 2020.

[19] A. Dalskov, D. Escudero, and M. Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security Symposium*, pages 2183–2200, 2021.

[20] 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.

[21] I. Damgård and J. Nielsen. Scalable and unconditionally secure multiparty computation. In *Advances in Cryptology – CRYPTO*, pages 572–590, 2007.

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

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

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

[25] J. Deng, W. Dong, R. Socher, L. Li, K Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[26] 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.

[27] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *Advances in Cryptology – CRYPTO*, pages 823–852, 2020.

[28] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. IACR Cryptology ePrint Archive Report 2020/338, 2020.

[29] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*, pages 291–326. 2022.

[30] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861, 2017.

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

[32] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2704–2713, 2018.

[33] 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.

[34] M. Keller. MP-SPDZ: A versatile framework for multi-party computation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1575–1590, 2020.

[35] M. Keller, D. Rotaru, N. P. Smart, and T. Wood. Reducing communication channels in MPC. In *International Conference on Security and Cryptography for Networks (SCN)*, pages 181–199, 2018.

[36] M. Keller and K. Sun. Secure quantized training for deep learning. In *International Conference on Machine Learning*, pages 10912–10938, 2022.

[37] 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.

[38] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. CrypTFlow: Secure tensorflow inference. In *IEEE Symposium on Security and Privacy*, pages 336–353, 2020.

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

[40] S. Li, K. Xue, B. Zhu, C. Ding, X. Gao, D. Wei, and T. Wan. FALCON: A fourier transform based approach for fast and secure convolutional neural network predictions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8705–8714, 2020.

[41] 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.

[42] 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.

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

[44] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In *International Conference on Financial Cryptography and Data Security*, pages 249–270, 2021.

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

[46] 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.

[47] 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.

[48] 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.

[49] A. Patra and A. Suresh. BLAZE: Blazing fast privacy-preserving machine learning. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[50] 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.

[51] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine

Protocol 5 $[a_1], \ldots, [a_m] \leftarrow \mathsf{Input}(a_1, \ldots, a_m)$

```
1:  for $T \in \mathcal{T} \setminus \{T^*\}$ do
2:      input owner generates random $k_T$ and sends it to each $p \in T$;
3:  end for
4:  for $i \in [1, m]$ do
5:      for $T \in \mathcal{T} \setminus \{T^*\}$ do
6:          each $p \notin T$ sets share $a_{i,T} = \mathsf{PRG}(k_T).\mathsf{next}$;
7:      end for
8:      input owner computes $a_{i,T^*} = a_i - \sum_{T \in \mathcal{T} \setminus \{T^*\}} \mathsf{PRG}(k_T).\mathsf{next}$ (in $\mathcal{R}$) and sends it to $p \notin T^*$;
9:      each $p \notin T^*$ sets share $a_{i,T^*}$ to the value received from input owner;
10: end for
11: return $[a_1], \ldots, [a_m]$;
```

learning applications. In *Asia Conference on Computer and Communications Security (ASIACCS)*, pages 707–721, 2018.

[52] D. Rotaru and T. Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In *INDOCRYPT*, pages 227–249, 2019.

[53] 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.

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

[55] S. Wagh, D. Gupta, and N. Chandran. SecureNN: 3-party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019(3):26–49, 2019.

[56] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2021(1):188–208, 2021.

[57] 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

A.1 Inputting Private Values

We start with a general case when a participant who is not a computational party supplies their input into the computation and consequently discuss an optimized version when the input owner is one of the computational parties. The input owner holds a private value a which will be represented as an element of ring $\mathcal{R}$. The input owner will need to generate replicated shares that correspond to a and send them to the computational parties. This will be the easiest way to proceed when there is only one element to share. However, when someone is sharing a vector of elements, we can save on communication by using pseudo-random shares. All shares except one for any element can be pseudo-random and computed locally by computational parties after obtaining a PRG seed. This means that among all shares $T \in \mathcal{T}$, one is marked as special and is denoted as T^*. The corresponding share is computed by the input owner and is communicated to all parties with access to that share. The construtcion is given as Protocol 5.

When the input owner is one of the computational parties, we can capitalize on the fact that the parties already have pre-distributed PRG seeds. We denote the input party as p^*. Note that p^* has access to a subset of the PRG seeds corresponding to the shares it is entitled to have access to, but not to all seeds. While we could generate new seeds for each T such that $p^* \in T$ and make it available to all $p \notin T$ and p^*, these seeds will be accessible to more than t parties and do not contribute to security. Therefore, we instead choose to set such shares to 0 and use only shares accessible to p^*. As a result, T^* will be such that $p^* \notin T^*$, the parties will set shares $a_T = \mathsf{PRG}(k_T).\mathsf{next}$ for each T such that $p \notin T$ and $T \neq T^*$, share T^* will be computed as $a_{T^*} = a - \sum_{T \text{ s.t. } p \notin T \wedge T \neq T^*} a_T$ (in $\mathcal{R}$) by p^* and communicated to all $p \notin T^*$, and all remaining shares a_T are set to 0.

All variants use a single round. When a single input is shared by an external party, the input owner simply generates all $\binom{n}{t}$ shares and communicates them to the computational parties (each share is stored by $t+1$ participants). This cost (which becomes sharing of a PRG seed) is amortized among all inputs when sharing multiple inputs. The additional cost per input for the input owner becomes generation $\binom{n}{t} - 1$ pseudorandom ring elements and communicating the last, computed share to $t+1$ computational parties, i.e., the total communication is $t+1$ ring elements. Each computational party needs to generate $\binom{n-1}{t}$ or $\binom{n-1}{t} - 1$ pseudo-random ring elements. When the input is shared by a computational party, there is no setup cost. The input owner need to generate $\binom{n-1}{t} - 1$ pseudo-random elements (i.e., similar to the number of shares it stores per shared value) and communicate the computed share to t other parties. Each other party computes $\binom{n-2}{t}$ (i.e., the number of shares it has in common with the data owner) or $\binom{n-2}{t} - 1$ pseudo-random ring elements. As will be relevant later, when a computational party is sharing a ring element in the (3,1) setting, the input owner communicates a single ring element to another party (and only one pseudo-random element is computed by the input owner and the remaining computational party). The security proof can be found in Appendix B.

Theorem 3. Input *is secure according to definition 1 in the* (n, t) *setting with* $n = 2t + 1$ *in the presence of secure communication channels and assuming* PRG *is a pseudo-random generator.*

A.2 Random Bit Generation

Random bit generation is a crucial component of a variety of protocols including different types of comparisons, bit decomposition, division, etc. Therefore, it is of paramount importance to support this functionality for general-purpose computation. In this work we examine two variants: (i) generating shares of a single bit as full-size ring elements and (ii) generating shares of k-bit random r as full-size ring elements together with generating shares of individual bits of r in $\mathbb{Z}_2$.

The first variant, denoted $\mathsf{RandBit}$, originated in [13] for field-based SS and was modified in [20] to work in $\mathbb{Z}_{2^k}$. We use the logic of [20] and adjust the algorithm to work in our setting. The result is shown as Protocol 6.

To achieve 50% probability of each outcome of the output bit, the computation uses a larger ring $\mathbb{Z}_{2^{k+2}}$ for most steps of the protocol when the remaining computation uses ring $\mathbb{Z}_{2^k}$. Consequently, 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 Protocol 6 follows from [20] and security follows from the logic. That is, because the protocol only discloses random e and otherwise uses secure building blocks, no information about private values can be leaked. The protocol runs in one round using

Protocol 6 $[b] \leftarrow \mathsf{RandBit}()$

```
1: $[u]_{k+2} \leftarrow \mathsf{PRandR}(k+2)$;
2: $[a]_{k+2} = 2[u]_{k+2} + 1$;
3: $e \leftarrow \mathsf{MulPub}([a]_{k+2}, [a]_{k+2})$;
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 $[b]_k$;
```

Protocol 7 $([r]_k, [b_0]_1, \ldots, [b_{k-1}]_1) \leftarrow \mathsf{edaBit}(k)$

```
1: **for** $p = 1, \ldots, t+1$ in parallel **do**
2:     party $p$ samples $r_0^{(p)}, \ldots, r_{k-1}^{(p)} \in \mathbb{Z}_2$ and computes $r^{(p)} = \sum_{j=0}^{k-1} r_j^{(p)} 2^j$;
3:     simultaneously execute $[r^{(p)}]_k \leftarrow \mathsf{Input}(r^{(p)}, k)$ and $[r_i^{(p)}]_1 \leftarrow \mathsf{Input}(r_i^{(p)}, 1)$ for $i = 1, \ldots, k$ with $p$ being the input owner;
4: **end for**
5: $[r]_k = \sum_{p=1}^{t+1} [r^{(p)}]_k$;
6: $s = t+1$;
7: **for** $i = 1, \ldots, \lceil \log(t+1) \rceil$ **do**
8:     **for** $j = 1, \ldots, \lfloor s/2 \rfloor$ in parallel **do**
9:         $\langle [r_1^{(j)}]_1, \ldots, [r_{k-1}^{(j)}]_1 \rangle \leftarrow \mathsf{BitAdd}(\langle [r_1^{(2j-1)}]_1, \ldots, [r_{k-1}^{(2j-1)}]_1 \rangle, \langle [r_1^{(2j)}]_1, \ldots, [r_{k-1}^{(2j)}]_1 \rangle)$;
10:        **if** $s \bmod 2 = 0$ **then**
11:            $s = s/2$;
12:        **else**
13:            $\langle [r_1^{((s+1)/2)}]_1, \ldots, [r_{k-1}^{((s+1)/2)}]_1 \rangle = \langle [r_1^{(s)}]_1, \ldots, [r_{k-1}^{(s)}]_1 \rangle$;
14:            $s = (s+1)/2$;
15:        **end if**
16:    **end for**
17: **end for**
18: $[b_0]_1, \ldots, [b_{k-1}]_1 = [r_0^{(1)}]_1, \ldots, [r_{k-1}^{(1)}]_1$
19: **return** $([r]_k, [b_0]_1, \ldots, [b_{k-1}]_1)$
```

the same communication as MulPub over $\mathbb{Z}_{2^{k+2}}$. To improve performance, in our implementation we compute the square root and inverse operations on line 4 simultaneously.

The second variant of random bit generation is based on the computation described in [27] and is denoted as edaBit(k), where the parameter k specifies the number of generated random bits as well as the bitlength of their representation as integer r. It produces secret-shared k-bit integer r together with shares of the individual bits of r in $\mathbb{Z}_2$. We use a simplified version with k being equal to the bitlength of the ring elements (i.e., the ring is $\mathbb{Z}_{2^k}$), which eliminates certain operations for dealing with carry after addition. The construction is given as Protocol 7. The idea consists of $t+1$ parties (without loss of generality, we chose the first $t+1$ parties for this role) each locally generating k random bits and computing representation of those bits as a k-bit integer (line 2). The bits are input into the computation using SS over $\mathbb{Z}_2$, while the integers are entered using shares in $\mathbb{Z}_{2^k}$ (line 3). Because we use Input to generate shares over different rings, we specify the second argument ℓ, which indicates that the shares need to be produced in $\mathbb{Z}_{2^\ell}$. The output that the protocol produces is the sum of the $t+1$ random integers (without the carry bits) and its bit decomposition is computed using bitwise addition BitAdd from [53] of the $t+1$ integers represented as bits in a tree-like manner.

Protocol	Rounds	Communication
RandBit()	1	$n-1$
edaBit(k)	$\log(t+1)(\log(k)+1)+1$	$t^2(\log(k)+1)+t+1/2$

Table 13: Performance of random bit generation protocols with communication measured in the number of ring elements sent per party over $\mathbb{Z}_{2^{k+2}}$ for RandBit **and $\mathbb{Z}_{2^k}$ for** edaBit(k)**.**

Protocol 8 $[a_{k-1}]_k \leftarrow \mathsf{MSB}([a]_k)$, where $a = \sum_{i=0}^{k-1} a_i 2^i \in \mathbb{Z}_{2^k}$

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

A.3 Comparisons

Less-than comparisons, $[a] < [b]$, are traditionally computed using SS by determining the most significant bit of the difference between a and b. Starting from [13], comparison protocols blind the difference by adding a random integer bit decomposition of which is known, open the sum, truncate all but one bit, and compensate for any carry caused by the addition. This logic was adapted to the ring setting in [20] by using building blocks that work over $\mathbb{Z}_{2^k}$. In the solution that we present as Protocol 8, we incorporate the edaBit protocol from [27] for efficient random bit generation into the construction of [20] adopted to the semi-honest setting. The presence of carry is determined using sub-protocol BitLT which performs comparison of two bit-decomposed values, one of which is given in the clear, using binary computation over $\mathbb{Z}_2$.

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 9) are information-theoretically protected using freshly generated randomness. The complexity of this protocol and its prior version that makes calls to RandBit is given in Table 3.

To correctly implement comparison 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 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.

B SECURITY DEFINITIONS AND PROOFS

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. Denote $\mathrm{VIEW}_\Pi(P_i)$ as 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: $\mathrm{VIEW}_\Pi(P_i) = (\mathsf{in}_i, r_i, m_1, \ldots, m_k)$. Let $I = \{P_{i_1}, P_{i_2}, \ldots, P_{i_t}\}$ denote a subset of the participants for $t < n$, $\mathrm{VIEW}_\Pi(I)$ denote 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. 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 simulator S_I such that $\{S_I(\mathsf{in}_I, f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)), f(\mathsf{in}_1, \ldots, \mathsf{in}_n)\} \equiv \{\mathrm{VIEW}_\Pi(I), (\mathsf{out}_1, \ldots, \mathsf{out}_n)\}$, where $\mathsf{in}_I = \bigcup_{P_i \in I}\{\mathsf{in}_i\}$ and $\equiv$ denotes computational or statistical indistinguishability.*

Proof of Theorem 1. 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 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 G_T.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. □

Proof of Theorem 2. 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 G_{T^*}.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 indistinguishability of the views. □

Proof of Theorem 3. It is straightforward to show security of the full version of Input when the input owner is different from the computational parties. That is, the input owner creates proper shares according to the SS scheme using a PRG. Thus, as long as security of the PRG holds, the real view is computationally indistinguishable from a simulated view created without the use of any secrets.

However, when the input owner is one of the computational parties, only a reduced set of shares is produced. Thus, we need to evaluate the combined view of each coalition of t corrupt participants. There are two important cases to consider: (i) input owner p^* is a part of the coalition and (ii) it is not.

When p^* is a corrupt participant, building a simulator is trivial: the simulator simply receives shares from the input owner on behalf of honest participants and terminates. Because inputs a_i are available to the corrupt parties, no information need to be protected and the real and simulated views use identical values.

When there are t corrupt participants who are different from p^*, we simulate the view by choosing a random value for a_{i,T^*} and sending it to each corrupt $p \notin T^*$. What remains to show is that the t corrupt parties do not possess enough shares to reconstruct the secret and, as a result, cannot learn any information about it. In more detail, p^* distributes its secrets using only shares T such that $T \in \mathcal{T} \setminus \{T^*\}$. However, because we use (n, t) threshold SS, there will be a share T possessed by p^* which is not available to any of the t corrupt parties I. Specifically that share is available to all participants except corrupt minority I. This means that the corrupt parties will not be able to reconstruct information about the private inputs and the real and simulated views are indistinguishable as long as PRG's security holds. □

C 5PC AND 7PC MULTIPLICATION MAPS

We define the necessary mappings for our multiplication protocol $[a] \cdot [b]$ (Protocol 1). Since ρ is substantially larger for 5 and 7 parties, we instead give one possible expression to calculate $v^{(p)}$ for an arbitrary party p. All index calculations are performed mod n.

For the 5-party configuration, we assign a unique index (i) to each $T \in S_p$ for party p:

$$\begin{aligned} (1) &= \{p+1, p+2\} & (4) &= \{p+2, p+3\} \\ (2) &= \{p+1, p+3\} & (5) &= \{p+2, p+4\} \\ (3) &= \{p+1, p+4\} & (6) &= \{p+3, p+4\} \end{aligned}$$

such that we use $a_{(i)}, b_{(i)}$ in place of a_{T_1}, b_{T_2} in the expression for $v^{(p)}$. Then, the product of all shares can be computed by party p:

$$\begin{aligned} v^{(p)} = {} & a_{(1)} \left(\sum_{i=1}^{6} b_{(i)} \right) + a_{(2)} \left(\sum_{i=1}^{6} b_{(i)} \right) + a_{(3)} \left(b_{(2)} + b_{(4)} \right) \\ & + a_{(4)} \left(b_{(1)} + b_{(3)} \right) + a_{(5)} \left(b_{(1)} + b_{(2)} \right) + a_{(6)} \left(b_{(1)} + b_{(5)} \right). \end{aligned}$$

Lastly, we define the mapping $\chi(p) = \{p+1, p+2\}$.

For the 7-party configuration, we once again assign a unique index (i) to each $T \in S_p$ for party p such that:

$$\begin{aligned} (1) &= \{p+1, p+2, p+3\} & (7) &= \{p+1, p+3, p+6\} \\ (2) &= \{p+1, p+2, p+4\} & (8) &= \{p+1, p+4, p+5\} \\ (3) &= \{p+1, p+2, p+5\} & (9) &= \{p+1, p+4, p+6\} \\ (4) &= \{p+1, p+2, p+6\} & (10) &= \{p+1, p+5, p+6\} \\ (5) &= \{p+1, p+3, p+4\} & (11) &= \{p+2, p+3, p+4\} \\ (6) &= \{p+1, p+3, p+5\} & (12) &= \{p+2, p+3, p+5\} \end{aligned}$$

$$\begin{aligned} (13) &= \{p+2, p+3, p+6\} & (17) &= \{p+3, p+4, p+5\} \\ (14) &= \{p+2, p+4, p+5\} & (18) &= \{p+3, p+4, p+6\} \\ (15) &= \{p+2, p+4, p+6\} & (19) &= \{p+3, p+5, p+6\} \\ (16) &= \{p+2, p+5, p+6\} & (20) &= \{p+4, p+5, p+6\} \end{aligned}$$

The product of all shares can be computed by party p:

$$\begin{aligned} v^{(p)} = {} & a_{(1)} \left(\sum_{i=1}^{20} b_{(i)} \right) + a_{(2)} \left(\sum_{i=1}^{20} b_{(i)} \right) + a_{(3)} \left(\sum_{i=1}^{20} b_{(i)} \right) + a_{(4)} \left(\sum_{i=1}^{20} b_{(i)} \right) \\ & + a_{(5)} \left(b_{(1)} + b_{(2)} + b_{(3)} + b_{(4)} + b_{(11)} + b_{(12)} + b_{(13)} + b_{(14)} + b_{(16)} \right) \\ & + a_{(6)} \left(\sum_{i=1}^{20} b_{(i)} \right) + a_{(7)} \left(b_{(3)} + b_{(6)} + b_{(8)} + b_{(10)} + b_{(12)} + b_{(14)} \right) \\ & + a_{(8)} \left(b_{(1)} + b_{(5)} + b_{(6)} + b_{(7)} + b_{(11)} + b_{(12)} + b_{(13)} \right) \\ & + a_{(9)} \left(b_{(1)} + b_{(5)} + b_{(6)} + b_{(11)} + b_{(12)} + b_{(17)} \right) \\ & + a_{(10)} \left(b_{(2)} + b_{(5)} + b_{(8)} + b_{(9)} + b_{(11)} + b_{(14)} \right) \\ & + a_{(11)} \left(b_{(1)} + b_{(2)} + b_{(3)} + b_{(4)} + b_{(5)} + b_{(6)} + b_{(10)} \right) \\ & + a_{(12)} \left(b_{(1)} + b_{(2)} + b_{(5)} + b_{(7)} + b_{(8)} + b_{(9)} \right) \\ & + a_{(13)} \left(b_{(1)} + b_{(2)} + b_{(3)} + b_{(5)} + b_{(6)} + b_{(8)} \right) \\ & + a_{(14)} \left(b_{(1)} + b_{(5)} + b_{(6)} + b_{(7)} \right) + a_{(15)} \left(b_{(3)} + b_{(5)} + b_{(6)} \right) \\ & + a_{(16)} \left(b_{(2)} + b_{(5)} \right) + a_{(17)} \left(b_{(1)} + b_{(2)} + b_{(3)} + b_{(4)} + b_{(16)} \right) \\ & + a_{(18)} \left(b_{(1)} + b_{(2)} + b_{(3)} \right) + a_{(19)} \left(b_{(2)} + b_{(9)} \right) \\ & + a_{(20)} \left(b_{(1)} + b_{(6)} + b_{(7)} \right). \end{aligned}$$

Lastly, we define the mapping $\chi(p) = \{p+1, p+2, p+3\}$.

D WAN MICRO-BENCHMARKS

The experimental configuration for WAN micro-benchmarks use the same servers as the LAN micro-benchmarks, with an added 23 ms one-way latency and the bandwidth throttled to 76 Mbps.

	Setup		Batch Size						
			1	10	10^2	10^3	10^4	10^5	10^6
3PC	$\mathbb{F}$G	30	23.5	23.5	23.7	25.9	40.9	291	2669
		60	23.5	23.6	23.7	26.3	70.8	438	4187
	$\mathcal{R}$	30	23.5	23.5	23.5	23.6	24.6	169	1525
		60	23.6	23.6	23.5	23.6	52.7	311	2994
5PC	$\mathbb{F}$G	30	23.5	23.5	23.7	27.1	49.7	448	4285
		60	23.5	23.5	23.8	27.4	79.6	724	6927
	$\mathbb{F}$D	30	47.1	47.1	47.2	52.3	91.2	518	4678
		60	47.1	47.1	47.3	52.8	115	758	6957
	$\mathcal{R}$	30	47.0	46.9	47.0	47.5	54.2	482	4934
		60	46.9	47.0	46.8	47.9	105	981	9443
7PC	$\mathbb{F}$G	30	23.5	23.5	23.9	28.6	85.6	661	6109
		60	23.5	23.6	24.1	29.1	116	1035	9843
	$\mathbb{F}$D	30	47.1	47.1	47.9	57.7	126	860	7513
		60	47.1	47.2	48.1	58.4	162	1235	10972
	$\mathcal{R}$	30	70.5	70.4	70.8	74.9	174	1484	11503
		60	70.4	70.4	70.7	76.1	253	2426	21083

Table 14: WAN runtime of multiplication protocols in ms. $\mathbb{F}$G and $\mathbb{F}$D refer to the optimized GRR and DN field multiplication from [11], resp., and $\mathcal{R}$ is our ring realization. 30 and 60 are integer bitlengths.

	Setup	Matrix Size			
		10×10	100×100	500×500	1000×1000
3PC	$\mathbb{F}$ (30)	23.8	116	1,278	9,462
	$\mathbb{F}$ (60)	23.8	119	1,556	10,292
	$\mathcal{R}$ (30)	23.3	28.3	582	2,842
	$\mathcal{R}$ (60)	23.3	57.2	968	4,489
5PC	$\mathbb{F}$G (30)	23.9	119	1,375	10,313
	$\mathbb{F}$G (60)	24.0	123	1,733	12,163
	$\mathbb{F}$D (30)	47.5	146	1,692	12,492
	$\mathbb{F}$D (60)	47.5	153	2,185	14,952
	$\mathcal{R}$ (30)	46.7	63.1	2,103	10,155
	$\mathcal{R}$ (60)	46.7	116	3,297	15,938
7PC	$\mathbb{F}$G (30)	24.3	121	1,483	11,296
	$\mathbb{F}$G (60)	24.3	126	1,845	12,923
	$\mathbb{F}$D (30)	48.2	147	1,886	15,389
	$\mathbb{F}$D (60)	48.2	155	2,443	16,598
	$\mathcal{R}$ (30)	71.1	206	8,200	58,142
	$\mathcal{R}$ (60)	71.3	301	12,018	98,145

Table 15: WAN runtime of matrix multiplication in ms.

	Setup	Batch Size						
		1	10	10^2	10^3	10^4	10^5	10^6
3PC	$\mathbb{F}$ (30)	23.5	23.5	24.1	28.4	58.8	495	4475
	$\mathbb{F}$ (60)	23.5	23.5	24.2	30.1	101	743	6870
	$\mathcal{R}$ (30)	23.6	23.5	23.5	23.9	32.8	232	2120
	$\mathcal{R}$ (60)	23.6	23.6	23.5	24.9	67.5	430	4173
5PC	$\mathbb{F}$ (30)	23.5	23.6	24.7	33.9	95.7	944	9157
	$\mathbb{F}$ (60)	23.5	23.6	24.9	35.1	142	1382	12781
	$\mathcal{R}$ (30)	23.5	23.5	23.6	24.3	41.6	386	3803
	$\mathcal{R}$ (60)	23.5	23.5	23.5	25.1	83.0	763	7493
7PC	$\mathbb{F}$ (30)	23.7	23.7	23.9	26.1	81.6	626	6201
	$\mathbb{F}$ (60)	23.7	23.7	23.9	27.0	123	1111	11105
	$\mathcal{R}$ (30)	23.6	23.6	23.8	25.6	79.7	627	6017
	$\mathcal{R}$ (60)	23.7	23.6	24.0	27.4	129	1194	11626

Table 16: WAN runtime of RandBit **protocols in ms.**

	Protocol	Batch Size						
		1	10	10^2	10^3	10^4	10^5	10^6
3PC	$\mathcal{R}$ (30)	164	165.079	164	169	233	1525	13899
	$\mathcal{R}$ (60)	189	188.855	188	198	414	3062	29212
	[3] (32)	1045	1044	1045	1046	1050	9834	97638
	[3] (64)	1849	1849	1850	1850	1859	17351	172588

Table 17: WAN runtime of edaBit **protocols in ms compared to MP-SPDZ implementation.**

	Protocol	Batch Size						
		1	10	10^2	10^3	10^4	10^5	10^6
3PC	$\mathbb{F}$ (30)	164	169	205	639	5114	47002	469621
	$\mathbb{F}$ (60)	166	176	308	1442	12611	120213	1313728
	$\mathcal{R}$+rB (30)	187	187	189	244	859	7492	71238
	$\mathcal{R}$+rB (60)	210	211	222	464	2893	26743	262235
	$\mathcal{R}$+eB (30)	351	351	351	357	483	2656	23065
	$\mathcal{R}$+eB (60)	398	398	398	410	849	5015	46611
	[3]+eB (32)	1448	1449	1448	1451	1491	11099	105248
	[3]+eB (64)	2253	2253	2254	2219	2398	18801	181247
	[3]+ABY3 (32)	452	453	500	1160	7839	74717	741118
	[3]+ABY3 (64)	476	477	524	1178	7891	75042	744067
5PC	$\mathbb{F}$ (30)	166	174	229	998	8533	79248	801954
	$\mathbb{F}$ (60)	168	184	364	2348	21007	202218	2119692
	$\mathcal{R}$+rB (30)	351	351	377	507	1636	14508	139451
	$\mathcal{R}$+rB (60)	398	398	528	863	5351	50290	493042
	$\mathcal{R}$+eB (30)	962	973	964	1003	1436	10662	98823
	$\mathcal{R}$+eB (60)	1103	1104	1110	1193	2805	23685	201298
7PC	$\mathbb{F}$ (30)	166	169	212	947	8512	82489	829618
	$\mathbb{F}$ (60)	167	175	371	2389	22649	222109	2239171
	$\mathcal{R}$+rB (30)	516	516	550	738	3181	26537	126586
	$\mathcal{R}$+rB (60)	586	588	737	1434	9297	84788	415354
	$\mathcal{R}$+eB (30)	1431	1432	1461	1682	6339	43072	368024
	$\mathcal{R}$+eB (60)	1642	1648	1710	2133	12181	85635	784269

Table 18: WAN runtime of MSB **protocols in ms unless marked otherwise. rB and eB indicate variants using RandBit and edaBit, respectively.**