PoPETs

Proceedings on Privacy Enhancing Technologies ; 2022 (4):1–27

Sameer Wagh

# Pika: Secure Computation using Function Secret Sharing over Rings

**Abstract:** Machine learning algorithms crucially depend on non-linear mathematical functions such as division (for normalization), exponentiation (for softmax and sigmoid), tanh (as an activation function), logarithm (for cross-entropy loss), and square root (for back-propagation of normalization layers). However, when machine learning is performed over secure computation, these protocols incur a large communication overhead and high round complexity. In this work, we propose new multi-party computation (MPC) protocols for such functions. Our protocols achieve constant round complexity (3 for semi-honest, 4 for malicious), an order of magnitude lower communication ($54-121\times$ lower than prior art), and high concrete efficiency ($2-1163\times$ faster runtime). We rely on recent advances in function secret sharing (FSS) to construct these protocols. Our contributions can be summarized as follows:

(1) A constant round protocol to securely evaluate non-linear functions such as division, exponentiation, logarithm, and tanh (in comparison to prior art which uses round complexity proportional to the rounds of iterative methods/required precision) with high accuracy. This construction largely follows prior work in look-up style secure computation.

(2) Our main contribution is the extension of the above protocol to be secure in the presence of malicious adversaries in the honest majority setting. We provide a malicious sketching protocol for FSS schemes that works over rings and in order to prove its security, we extend (and prove) a corresponding form of Schwartz-Zippel lemma over rings. This is the first such extension of the lemma and it can be of independent interest in other domains of secure computation.

(3) We implement our protocol and showcase order of magnitude improvements in runtime and communication. Given the low round complexity and substantially lower communication, our protocols achieve even better performance over network constrained environments such as WAN. Finally, we showcase how such functions can lead to scalability in machine learning.

Note that techniques presented are applicable beyond the application of machine learning as the protocols effectively present an efficient 1-out-of-N oblivious transfer or an efficient private information retrieval protocol.

**Keywords:** Multi-party computation, Function Secret Sharing, Distributed Point Function, Schwartz-Zippel Lemma over Rings

DOI 10.2478/popets-2022-0095
Received 2022-02-28; revised 2022-06-15; accepted 2022-06-16.

## 1 Introduction

Secure Multi-party Computation (SMC/MPC) is becoming an increasingly popular way to perform computation over data while maintaining privacy. Applications such as machine learning are uniquely positioned to benefit from access to more data while remaining privacy compliant. Such ML applications have been the focus of a number of recent works [31, 41, 59, 63, 64, 72, 86, 88]. While plaintext computation provides no privacy, MPC techniques introduce privacy into the computation at an overhead. Over the past few years, advances in protocol design coupled with systems improvements have led to reductions in the overhead of secure computation protocols.

Machine learning (ML) algorithms require both linear and non-linear computation. In plaintext computation, the non-linear operations are extremely efficient and most of the efforts are geared towards improving the efficiency of the linear layers. However, the balance of the overhead of the linear and non-linear layers is different under linear secret sharing-based secure computation - the non-linear layers are more expensive, requiring large communication and high round complexity. To circumvent these drawbacks, non-linear layers are usually replaced using "MPC friendly" functions.

However, ML algorithms crucially rely on non-linear functions. Normalization is essential for stable convergence. Sigmoid and logarithm (for cross-entropy) are necessary to achieve high learning accuracy. Inverse square root is critical to enable backward propagation over normalization layers and thus fundamental to training neural networks. Tanh enables trained models to achieve much better performance on certain machine learning tasks such as natural language processing. Non-linear functions are thus essential for improved learning, better convergence, and high

**Sameer Wagh:** Devron Corporation and UC Berkeley, E-mail: swagh@berkeley.edu

accuracy [51, 52] and consequently are a fundamental component of the machine learning ecosystem.

Even though these non-linear functions are essential, they incur large overheads when implemented in secure computation [21, 22, 54, 71, 88]. Furthermore, for more complex non-linear functions such as exponentiation and logarithm, the set of techniques is restricted to either some form of iterative methods or Taylor series expansions. These require running the iterative methods until sufficiently good convergence has been achieved. These factors make the use of such non-linear functions prohibitively expensive within secure computation. Additionally, secure computation frameworks for machine learning algorithms typically use fixed-point arithmetic. This makes the computations, particularly multiplications, inherently approximate. Non-linear functions, which are typically computed using polynomial approximations, also rely on sequential multiplications and thus incur further approximation in their computation. When these errors accumulate over computations of high depth, the overall accuracy of the computations is diminished. Thus, without the use of functions such as sigmoid or integer division (as normalization), scaling such computations to larger depths is extremely difficult. To truly enable secure machine learning over large scales, support for highly efficient protocols to compute non-linear functions is the need of the hour.

In this work, we design highly efficient protocols for non-linear functions such as normalization, exponentiation, logarithm, and square root. Our protocols can be used for both private training of ML models as well as for private inference (where the model may be trained elsewhere). Our protocols exhibit constant round complexity in contrast to prior work that requires round complexity proportional to the number of iterations of the Newton's method. We focus on MPC over finite rings and design protocols for both semi-honest and malicious security with high concrete efficiency (three non-colluding servers).

## 1.1 Our Contributions

Below, we describe our main contributions in detail.

**Constant Round Protocols for Non-linear Functions.** We demonstrate constant round protocols for functions such as normalization, exponentiation, logarithm, square root, tanh etc – functions that are critical to present day machine learning algorithms and frameworks. In particular we ask the following question:

> Can we design constant round protocols for non-linear functions that achieve high concrete efficiency when considering MPC over finite rings?

and answer it affirmatively by constructing protocols to compute these functions within MPC. To achieve this goal, we use Function Secret Sharing (FSS), a primitive introduced by Boyle *et al.* [15] at Eurocrypt 2015. FSS provides a way for additively secret sharing a function that can be evaluated locally. We adopt a look-up style approach [36, 42, 55] to show that in the setting of machine learning applications, our FSS based protocols achieve orders of magnitude better runtime and require substantially less communication. While recent works such as [9] share our goals, they rely on a fundamentally different usage of FSS schemes. In particular, these constructions still require iterative methods to compute the functions and thus do not enjoy a truly constant round complexity.

**Malicious Sketching Scheme over Rings.** Our main contribution is the extension of our semi-honest protocol to achieve malicious security with abort in the presence of an honest majority corruption. The challenge in this extension is to ensure the FSS keys are well formed. These techniques are known as sketching schemes in the literature ([16] Section 4). They use the Schwartz-Zippel lemma [79, 92] (refer to Lemma 5.1 in Section 5 for an informal statement and to Appendix G.2 for a historical overview of the lemma) to ensure that malicious behavior escapes detection with a probability inversely proportional to the size of the finite field. In our work, to achieve high concrete efficiency, we focus on MPC over finite rings. This is a problem as the Schwartz-Zippel lemma (SZ) does not hold over rings. A polynomial over a finite ring can have many more roots than its degree (this is the basis for the SZ lemma). For instance, the polynomial $p(x) = 2^{\ell-1}x$ over $\mathbb{Z}_{2^\ell}$ is a linear polynomial but every even value is a root. To address this, we state (and prove) a new lemma that enables us to achieve efficient malicious sketching over rings. The formal statement is presented in Lemma 5.4 and is informally stated below:

**Lemma 1.1.** *(**Special Form of Schwartz-Zippel lemma over Rings (Informal)**) Let $p(x_1,\dots,x_n)$ be a $n$-variable bilinear polynomial over $\mathbb{Z}_{2^{\ell+2s}}$ that is not identically zero over $\mathbb{Z}_{2^\ell}$ The probability that $p$ evaluates to 0 over randomly sampled points from $\{0,1,...,2^s-1\}$ is bounded above by $2^{1-s}$.*

Note that all prior works that use malicious sketching routines [8, 16, 28, 45] use finite fields to perform the malicious sketching. Thus, to the best of our knowledge, this is the first time such a result has been extended to hold over rings. Secure computation over finite rings is increasingly being used due to the ease of implementation and improved performance [26, 29, 32, 64, 67, 88]. We discuss other extensions of the SZ lemma in Section 7 and Appendix G.

**Implementation and Concrete Efficiency.** We implement and benchmark our protocol against state-of-the-art implementations from ABY$^3$ [63] and MP-SPDZ [38]. We also compare against a state-of-the-art 2PC protocol from SiRnn [71] and the size revealing but efficient 3PC protocol from Falcon [88] and show impressive gains. Our protocols achieve up to 128× higher throughput while using 23× lower communication compared to ABY$^3$ for smaller batch sizes. When compared to the more versatile codebase MP-SPDZ, our protocols achieve up to 25× higher throughput using 74× lower communication.

Where our protocols shine even more is when considering a network constrained environment such as WAN. For instance, over a 100 Mbps network link, our protocols provide an impressive 770× improvement compared to MP-SDPZ - while our computational costs are dominant, the protocols consume $100-200\times$ less communication and over just 3 rounds (4 for malicious; 2 rounds if considering online only). The dominant cost of our protocol comes from the full domain EvalAll and the inner product required for the look-up (refer to Appendix D for these building blocks). While we leverage parallelism and a few other optimizations in our evaluation (refer to Section 6 for optimizations), we expect that additional optimizations such as more efficient implementations of the inner product will further improve the performance of our protocols.

Finally, we also run important experiments over neural networks to showcase how our protocols can pave the way for scaling neural networks to larger depths. We show that the errors due to the approximate nature of MPC computation accumulate over a large depth (as low as 5 or 6 layers) and soon start affecting the overall performance of the computation. In order to scale neural networks, avoiding overflow in the MPC computation and preventing the snowballing of errors is of utmost importance. The known techniques ML experts use in order to avoid this problem are normalizations layers (or functions such as sigmoid that convert vectors into probability distributions). We show that the use of normalizations not only improves the training accuracy and convergence (already known to ML experts) but also that the relative error remains at an acceptable constant thus allowing for arbitrary depth computations. This is critical for scaling deep learning.

# 2 Background

In this section, we describe some important techniques and concepts that are critical to this work.

## 2.1 Machine Learning Using MPC

Applications of secure computation, in particular machine learning, heavily rely on the ability to compute over rational numbers. Designing secure computation protocols over rational numbers has been a known challenge. Rational numbers are implemented using fixed-point representation and the precision is set according to the application in consideration.

Support for non-linear functions is critical in secure computation for two reasons. Firstly, non-linear functions such as normalization, sigmoid, tanh are commonplace and crucial in present day machine learning applications. They enable improved learning (by using non-linear activation functions such as ReLU and tanh) and provide better convergence (by using batch normalization and appropriate loss functions). While approximate techniques are shown to work in certain applications over standard datasets such as MNIST, the results do not always generalize [39]. Thus, it is important to support high precision non-linear function computation.

Secondly, secure computation necessitates support for such functionalities. When MPC protocols are implemented, the actual values involved in computations can be integers or floating-point values (converted into fixed-point values) and are assumed to lie within some integer range $(-2^{k-1},2^{k-1}]$ (refer to Section 3 for details). When computation is performed over large depths, the values increase in size and may overflow such a range. In this case, MPC output loses its correctness. MPC frameworks deal with this issue by having the MPC user ensure that the previously mentioned constraints are met (for example, Section 14.8.1 in [1]). When the computation is large and the values are unknown, it is hard to know that the invariant is broken. This is where non-linear functions are extremely important. Functions such as normalization, sigmoid, and tanh enable mapping inputs from an arbitrary range to a fixed bounded range. This allows for arbitrary depth computations without worrying about overflows and thus is extremely critical to scaling secure computation.

**Techniques for Normalization, Exponentiation.** Prior work decomposes the above set of non-linear functions into two basic primitives: normalization (integer division) and exponentiation. Other functions can then be computed as a composition of these and other basic MPC primitives. Given an input $x$ (say as a fixed-point number), normalization deals with computing $\lfloor 1/x \rceil$ (the nearest integer) where 1 is interpreted as a fixed-point value. Protocols for these are usually adaptations of the seminal work in [21, 22]. The protocols typically come in two flavors – approaches based on piece-wise linear computation [9, 64, 80] and approaches based on iterative methods (Newtons methods or series expansion) [22, 54, 88]. The former are better suited for computations with fewer pieces while the latter enjoy fast convergence and leverage "MPC friendly" computation. These functions all follow a similar structure and differ in how they implement each step:

(1) Start with an initial approximation.
(2) Recast the input to an appropriate range.
(3) Finally, run the iterative methods for a certain number of rounds till sufficient convergence is achieved.

Note that because of their structure, these techniques incur a round complexity that is dependent on the precision desired. Furthermore, such approaches provide approximate computation for two reasons - the first is the approximation involved in the MPC computation itself (such as multiplication/polynomial evaluations over fixed-point) and the second is the function itself is approximated to a polynomial expression due to the use of iterative methods.

**Other Applications.** While the focus of this work is to provide lightweight cryptographic protocols for machine learning applications, these primitives (normalization, sigmoid, tanh, logarithm etc.) are also required in other applications of secure computation such as secure clustering [18, 43], secure auctions [65], genomics computation [4], and anti-money laundering [77].

## 2.2 Function Secret Sharing

Function Secret Sharing (FSS), a primitive introduced by Boyle *et al.* [15] at Eurocrypt 2015, provides a way for additively secret sharing a function that can be evaluated locally. Thus, given a function $f$ (more generally a family of functions), $f$ can be split into $m$ succinctly described functions $f_1,...,f_m$ such that any strict subset of $\{f_1,...,f_m\}$ hides $f$ but $f = f_1 + \cdots + f_m$ at each point within the domain. FSS is motivated by two types of applications: those that require privately reading from a database and those that require privately writing to a database. In both these cases, FSS can be used to significantly reduce the communication overhead.

An example of a database reading application is a primitive known as private information retrieval (PIR). In this case, the problem of private database reading can be described as follows: A number of servers (say 2 servers) hold a database $D \in \mathbb{F}_2^\ell$ and a client would like to retrieve the element $d_k$ without revealing the index $k$. The central ideas can be traced back to the seminal work by Chor *et al.* [27]. The client generates two bit vectors $y_0,y_1$, each of the size of the database, such that

$$y_0 \oplus y_1 = e_k \tag{1}$$

where $e_k$ is the unit vector with 1 at the index $k$. The client sends the vector $y_\sigma$ to server $S_\sigma$ for $\sigma = 0,1$ and the server responds with the inner product of the entries of the database with the vector $y_\sigma$. The client can XOR the two received values to extract the database entry of interest (this requires that the database be considered over a field of characteristic 2). Note that the vectors $y_0,y_1$ are linear in $N$, the size of the database. This makes their communication linear in the size of the database. FSS is ideally suited to address this problem providing an elegant construction for expanding $y_0,y_1$ from shorter seeds $k_0,k_1$, thus significantly reducing the communication footprint of the protocol. This is because $y_0,y_1$ are shares of a function (distributed point function, see below) for which efficient FSS constructions are known in practice.

As an example of a solution to problems concerning privately writing to a database, consider Oblivious RAMs [49, 50]. This primitive considers reading and writing from an outsourced database. FSS has also shown promise in improving Oblivious RAM schemes by (1) reducing communication and (2) scaling to larger database sizes [42]. Furthermore, a long line of work [8, 9, 11, 12, 17, 71] has explored the use of FSS in constructing efficient protocols for other MPC primitives.

**Distributed Point Functions.** Distributed Point Functions (DPFs) are a class of functions that evaluate to a nonzero value at exactly one index in their domain [15, 16, 47]. More formally, such a function can be defined as

$$f_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } \; x = \alpha \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Over the past few years, highly efficient FSS constructions have been proposed for this class of functions. The succinct size of the FSS keys allows secret sharing a "one-hot" vector of size $N = 2^n$ with only $O(\log N)$ bits of information. This property is useful in designing protocols for PIR-style applications [27]. More formally, a FSS scheme for the DPF function $f_{\alpha,\beta}(x)$ where the range is over a group $\mathbb{G}$ (which may be a ring or field) is defined by a pair of algorithms $(\mathsf{Gen},\mathsf{Eval})$ where:

– $\mathsf{Gen}(\alpha,\beta) \rightarrow (k_0,k_1)$ takes as input the location $\alpha \in \{0,1\}^n$ and the value $\beta \in \mathbb{G}$ of the function when evaluated at $\alpha$ and outputs two FSS keys (one per party).
– $\mathsf{Eval}(\sigma,k_\sigma,i) \rightarrow \mathbb{G}$ takes as input an FSS key and the location at which the function is to be evaluated and outputs the share of the $f_{\alpha,\beta}(i)$.

The correctness of the FSS construction ensures that $f_{\alpha,\beta}(i) = \mathsf{Eval}(0,k_0,i) + \mathsf{Eval}(1,k_1,i)$ where $(k_0,k_1) \leftarrow \mathsf{Gen}(\alpha,\beta)$. Similarly, the security of the DPF implies that an adversary who learns one of $k_0$ or $k_1$ but not both learns nothing about $\alpha$ or $\beta$. We also use a subroutine $\mathsf{EvalAll}$ that is simply the algorithm $\mathsf{Eval}$ computed at each location $i \in [N]$. For formal definitions and more details, we refer the reader to [15, 16, 47].

# 3 Technical Overview

Multi-party computation protocols come in a variety of flavors - from garbled circuits based to arithmetic or boolean secret sharing based. Our work is inspired by ideas that perform a look-up style computation [36, 55]. In fact, all garbled circuit based protocols can be considered since the Garbler sends the table, and the Evaluator evaluates the table at the function inputs. These approaches enjoy the benefit that the protocols are oblivious to the function as they simply involve "looking-up" the function value.

In a seminal work by Doerner and shelat [42], this idea of look-up computation is used to construct a 2-party Distributed Oblivious RAM scheme with high concrete efficiency. The idea there is to perform the look-up in a database using a secret shared index using a Chor style protocol (described in Section 2.2). The important component of Chor's protocol is the generation of two vectors that differ in the index of interest. However, these are exactly the outputs of a distributed point function. With the advances in function secret sharing, particularly efficient FSS constructions for a DPF [16], two vectors differing in one element can be efficiently generated within MPC. It is precisely in this blueprint that we construct our protocols. However, we encounter the following two challenges when naively constructing such protocols for non-linear functions:

(1) Most MPC frameworks require a representation of at least 32 bits. Doing a look-up table of this size is known to be impractical (typical sizes for which FSS provides performance comparable to general purpose MPC are around 20-25 bits [16, 42]).
(2) We focus on MPC over rings (with modulus $\mathbb{Z}_{2^\ell}$) and thus require the output of the FSS scheme to be over such a group. However, most known techniques to check for ill-formed FSS keys require finite fields [8, 16].

We solve the first challenge by leveraging the inherent structure of MPC implementations. The second challenge is overcome by proving an extension of Schwartz-Zippel lemma over finite rings. We briefly describe each of these insights in further detail below.

**Challenge 1: About MPC implementations.** When MPC protocols are implemented, the actual values to be computed (called *data*) can be integers or floating point values. However, in order to utilize cryptography and achieve strong security, we require the values used to encode this data when performing secure computation (called the *representation*) to be within a structured algebraic set such as a finite ring or field. The encoding of the data into the representation is one of the design choices of secure computation. Secure computation over finite rings such as $\mathbb{Z}_{2^\ell}$ has the advantage that the modular arithmetic occurs without any additional checks. Thus, compared to the use of general-purpose number theoretic libraries, performance significantly improves as the size of the computation scales. When considering the implementation of MPC protocols over finite rings, integers are naturally encoded into native data types such as `uint32_t` or `uint64_t`. Floating point values are encoded using a technique known as fixed-point encoding where each floating point value $x$ is converted into an integer $\lfloor x \cdot 2^f \rfloor$ where $f$ is the fixed-point precision. Typical values of the floating precision $f$ used in literature [64, 88] are about 9 to 17 bits - which translates to 3 to 6 decimal digits. Thus, the lower $f$ bits encode the fractional part and the higher bits encode the integer part.

Now suppose we consider an exponential function $g(x)=2^x$ to be computed within secure computation (such a function with base $e$ is required for sigmoid, tanh). If we use a 64-bit data type for the representation of the data, we know that the largest value we can represent is $2^{64}$ (ignoring the sign and the fixed-point precision). Thus, any data whose integer value exceeds 64, i.e., is more than 6-bits, would inherently impede any secure computation protocol. Most MPC frameworks work around this by assuming a bound on the inputs [1, 37, 54, 64, 88]. Thus for the exponential function, the inputs have to be less than 64. When translated into the fixed-point values, the representation is assumed to be bounded by $f+7$ bits (one additional bit for the sign). For the typical values of the floating precision used in state of the art frameworks, $f+7 \approx 16-24$ bits.

We use this observation to our advantage. We leverage such bounds on the input to quantize/reduce them into a smaller domain. The look-up is then performed using the FSS schemes for DPFs. Thus, we identify and use the constraints of MPC data representations as a feature in our protocols to achieve low round complexity and high concrete efficiency protocols. In order to scale computations to large depths, the above constraints on the bounds have to be maintained. This further motivates the need for normalization protocols, which is the focus of this work.

**Challenge 2: Malicious Security for FSS over Rings.** State-of-the-art FSS schemes [16] crucially rely on fields of characteristic 2. Thus, operations have to be performed over finite fields such as $\mathbb{F}_2$,$\mathbb{F}_{2^\ell}$. However, when considering the broader MPC implementation, there is a significant overhead to converting data representations between $\mathbb{F}_{2^\ell}$ and $\mathbb{Z}_{2^\ell}$. Thus it is important to ensure that the inputs and outputs of the FSS scheme are elements of the finite ring over which computation is to be performed.

Given the structure of our protocol, we use the party $P_2$ to generate the FSS keys that the parties $P_0$ and $P_1$ use in their computation. Designing an MPC protocol over a finite ring such as $\mathbb{Z}_{2^\ell}$ requires an intricate protocol design, particularly when considering malicious security[1]. In this setting, we have to defend against two types of corruptions:

(1) Party $P_2$ being malicious. In this case, the correlated randomness provided could be ill-formed and needs to be verified.
(2) One of party $P_0$,$P_1$ is malicious. In this case, the entire MPC computation has to be performed in a "dishonest majority style" computation given our asymmetric protocol design.

Our insight here is to solve both these problems at once using a SPD$\mathbb{Z}_{2^k}$ style data representation [29]. In this set-up each secret value is secret shared between $P_0$,$P_1$ along with a MAC on that secret (refer to Section 5 for more details). Thus in our set-up, we require that the party $P_2$ generate not just the FSS keys for the DPF shares but also for shares of the MAC on the DPF. The interesting observation here is that since the original shares are of a DPF, i.e., Hamming weight one, the MAC is also a DPF and can thus be represented using a DPF key with a different payload. While this solves the second corruption difficulty, we still need to address the first one.

To provide checks against ill-formed keys, we use the blueprint from [16] for malicious sketching. However, due to the requirement of both the DPF and the MAC DPF, the Schwartz-Zippel lemma does not provide any meaningful security. This is because the SZ lemma crucially relies on operations over a finite field. Simply considering the output groups as a subgroup of a larger field does not work when dealing with malicious servers. To address this, we prove a version of the SZ lemma over rings (Lemma 5.4) and use that to detect ill-formed FSS keys with a given statistical security. Security can be argued in the arithmetic black box model in both the semi-honest and malicious settings [35].

**Limitations of our Approach.** We briefly discuss the limitations of our protocols which will assist in determining which scenarios are best suited for our protocols. The bottleneck of our constructions is the full domain evaluation of the FSS scheme and thus our protocols provide best concrete efficiency where values are bounded within smaller ranges or can be quantized into smaller ranges. Furthermore, our protocols achieve constant round complexity and thus provide the most impressive gains when considering deployments in network constrained environments such as wide area network (WAN). Finally, the structure of our protocols is asymmetric and thus the distribution of load is unequal between the parties ($P_2$ performs different computation than $P_0$,$P_1$; however being entirely pre-computable, it may actually be a useful feature in certain applications). In cases where an evenly distributed workload is required, symmetric protocols such as [2, 46, 63, 88] would be more desirable.

# 4 Semi-honest Secure Protocol

In this section, we present a protocol that is secure against a semi-honest adversary. The protocol forms a basis for the maliciously secure protocol presented in Section 5.

## 4.1 Protocol Overview

We first describe the secret sharing set-up and then proceed to the protocol construction ($\Pi_{\mathsf{sh:Func}}$). Notice that our protocol description is oblivious to the function being computed (normalization, exponentiation, square root, trigonometric functions).

**Set-up and Notation.** The secret sharing set-up is as follows: we consider all the data to be 2-out-of-2 secret shared between $P_0,P_1$ and the modulus is $2^\ell$. The party $P_2$ does not hold shares of the data but will be used to generate common randomness required for the function computation. The function computation is parametrized by $f$, the fixed-point precision used for that function computation, and is denoted by $\mathsf{Func}$. The data representation is bounded within the range $(-2^{k-1},2^{k-1}]$ for a fixed constant $k$ depending on the application. Thus, for any plaintext value $a_{\mathsf{float}}$, the data is encoded as secret shares

$$a_{\mathsf{repr}} = a_0 + a_1 \pmod{2^\ell} \tag{3}$$

Where $a_{\mathsf{repr}}$ is the nearest integer to $a_{\mathsf{float}} \cdot 2^f$ and $a_\sigma \in \mathbb{Z}_{2^\ell}$ is held by party $P_\sigma$ for $\sigma = 0,1$. This set-up mimics prior works such as [76, 86]. Furthermore, we assume that unsigned integer data types are used and thus a negative value of $-1$ will be the element of $\mathbb{Z}_{2^\ell}$ consisting of all 1's. For the rest of the paper, we will drop the subscript of $a_{\mathsf{repr}}$ and use $[a]$ to denote the secret sharing with the implicit understanding that it refers to the representation. For malicious security, we use the technique of Message Authentication Code (MAC) from the SPDZ-line of work [29, 34, 37]. The details of these are presented in Section 5.1.

**Protocol Intuition.** The protocol is formally described in Fig. 1. Our protocol starts with sharing of the inputs $[a]_{2^\ell}$ and outputs a sharing $[b]_{2^\ell}$, where $b = \mathsf{Func}(a)$ for some publicly computable function $\mathsf{Func}$. Examples of such a function include (1) reciprocal computation, where $b = 1/a$ computed within fixed-point arithmetic of precision $f$, (2) sigmoid, where $b = 1/(1+e^{-a})$, once again computed as

[1] Once again note that naively considering the ring elements as an additive subgroup of a larger field does not work for our setting.

fixed-point value with precision $f$, or (3) square root, where $b=\sqrt{a}$ with fixed-point precision $f$.

First, the parties $P_0,P_1$ locally construct a database of the function values, i.e., a database $D_{\mathsf{Func}}$ with entries $d_i$ for $i\in 2^k$ such that $d_i=\mathsf{Func}(i)$ where the input $i$ is also considered as a fixed-point value (thus the database is a function of the fixed-point precision $f$). The problem of computing the function on the secret input $[a]_{2^\ell}$ then translates into the problem of database look-up. However, given that the shares are in $2^\ell$, we need to reduce them into shares over a smaller ring. Here we use the structure of the data encoding to reduce the input $[a]$ into $2^k$ given the bounds on the input. Then we use the PIR protocol (described in Section 2.2) to perform a look-up on the database.

To perform the PIR protocol, we need access to two vectors $u_0,u_1$ that differ only at the location $a$. However, since $a$ is secret, it cannot be revealed to any party. This is where we can make use of the party $P_2$ to generate correlated randomness. Party $P_2$ simply samples a random value $r\in 2^k$ and generates FSS keys with shares of the DPF corresponding to the value 1 at location $r$. Note that we can optimize the protocol to generate the output over $\mathbb{F}_2$ (cf Section 2.2) and thus avoid evaluating the last few layers of the FSS GGM tree (the last few layers contain the largest overhead of the evaluation [42]). However, since we want to support computation natively over $\mathbb{Z}_{2^\ell}$, we need to convert the Boolean shares into shares over $\mathbb{Z}_{2^\ell}$ which requires interaction. We avoid this by having each party locally convert Boolean values into $\mathbb{Z}_{2^\ell}$. However, as a consequence, the $\mathbb{Z}_{2^\ell}$ shares can be shares of +1 or -1. We correct for this issue by having party $P_2$ also generate shares of the "sign" of the DPF (this bit is denoted by $w$).

*Sign of the Distributed Point Function.* The output of the DPF is two vectors $y_0,y_1$ of size $N=2^k$ such that the following constraint holds:

$$y_0[i]\oplus y_1[i]=\begin{cases}1 & \text{if } \ i=r\\ 0 & \text{otherwise}\end{cases}$$

Thus, $y_0[i],y_1[i]\in\mathbb{F}_2$ such that $y_0[i]\oplus y_1[i]=\delta_{i,r}$ where $\delta_{i,j}$ is the Kronecker delta function. However, when these shares are naively lifted into $\mathbb{Z}_{2^\ell}$, then $y_0[i]+y_1[i]$ can be 0, 1, or 2 modulo $2^\ell$. To address this, we generate the FSS keys for a subtractive DPF, i.e., the vectors $y_0,y_1$ when expanded from the FSS keys satisfy the following constraint $y_0[i]-y_1[i]=\delta_{i,r}$. This modification requires no changes to the FSS construction because the conditions $y_0[i]\oplus y_1[i]$ and $y_0[i]-y_1[i]$ are equivalent over $\mathbb{F}_2$. However, when we perform the same naive lifting over $y_0,y_1$ into $\mathbb{Z}_{2^\ell}$, $y_0[i]-y_1[i]$ takes on the values 1 or $-1$ if $i=r$ and 0 otherwise. Thus, along with the FSS keys we also require party $P_2$ to send over shares of this sign value, i.e., shares of $\pm1$ depending on the FSS keys. Note that this sign can be read off, with no additional cost, from the FSS $\mathsf{Gen}$ routine by simply examining the value of the variable $t_0^{(\nu)}$ (refer to Fig. 7).

**Semi-honest protocol for function computation $\Pi_{\mathsf{sh:Func}}$**

**Auxiliary variables:** Parameters $\ell,k,f$ for bit-size, bound and fixed-point precision of inputs. The function to be computed $\mathsf{Func}$ and the function database $D_{\mathsf{Func}}$ of size $2^k$ with each entry $d_i$ in $\mathbb{Z}_{2^\ell}$ equal to $\mathsf{Func}(i)$ for $i\in[2^k]$.
**Inputs:** Secret shared values $[a]_{2^\ell}$.
**Outputs:** Secret shares $[b]_{2^\ell}$ where $b=\mathsf{Func}(a)$.

**Protocol:**

(0) Parties $P_0,P_2$ and $P_1,P_2$ invoke $\mathcal{F}_{\mathsf{Rand}}$ to generate values $r_0,r_1\in\mathbb{Z}_{2^\ell}$ such that $r_0$ is known to $P_0,P_2$ and $r_1$ is known to $P_1,P_2$. This step is non-interactive.

(1) Party $P_2$ generates the following correlated randomness:
  (a) Let $f_r$ be the single-bit DPF at $r$.
  (b) Generate $(k_0,k_1)\leftarrow\mathsf{Gen}(r,1)$ for the DPF $f_r$ (Figure 7).
  (c) Set $w=1$ if $t_0^{(\nu)}=1$ (final layer control bit, refer Figure 7) and $w=-1$ otherwise. Generates shares $(w_0,w_1)\in\mathbb{Z}_{2^\ell}^2$ of $w$ and send the tuple $(k_\sigma,w_\sigma)$ to party $P_\sigma$ for $\sigma\in\{0,1\}$.

(2) Party $P_\sigma$ for $\sigma\in\{0,1\}$ evaluates the following:
  (a) Reconstruct $x\equiv r-a \pmod{2^k}$
  (b) Compute $y_\sigma\leftarrow\mathsf{EvalAll}(\sigma,k_\sigma)$ (refer Fig. 7) where the components of $y$ are $\{y_\sigma[0],y_\sigma[1],...,y_\sigma[2^k-1]\}$
  (c) Set $u_\sigma[i]=y_\sigma[i+x]$ for $i\in\{0,1,...,2^k-1\}$ and compute $v=(-1)^\sigma\langle u_\sigma,D_{\mathsf{Func}}\rangle$, i.e.,

$$v_\sigma=(-1)^\sigma\sum_{i=0}^{2^k-1}u_\sigma[i]\cdot d_i$$

  where the product between a bit $\{0,1\}$ with an element of $\mathbb{Z}_{2^\ell}$ is performed by static casting $\{0,1\}\to\mathbb{Z}_{2^\ell}$.
  (d) Invoke $\mathcal{F}_{\mathsf{Rand}}$ to generate a random value $\beta$ that is used to re-randomize $v$: $v\leftarrow v+(-1)^\sigma\beta$

(3) Parties $P_0,P_1$ output $[c]_{2^\ell}=[v\cdot w]_{2^\ell}$ using a Beaver triple.

**Fig. 1.** Protocol for *semi-honest secure* computation of a given function $\mathsf{Func}$ (such as division, sigmoid, exponentiation, tanh).

*Function look-up.* The final component of the protocol is to use this randomness to compute the function look-up on the input $[a]$. We use a standard masking trick to achieve this. Party $P_2$ supplies arithmetic shares of $r$ (the index of the non-zero DPF value) to parties $P_0,P_1$ who then use it to reveal the value of $x\equiv r-a \pmod{2^k}$. Note that due to the fixed-point encoding, the database $D_{\mathsf{Func}}$ encodes the function computation on negative values in the lower half of the table which is consistent with the desired function computation. Finally, the output of the full domain evaluation $y_\sigma$ is appropriately shifted by $x$ (modulo $2^k$) and an inner product of the database is computed with this shifted vector[2]. Since the PIR is performed using subtractive shares, party $P_1$ multiplies the output of its inner product by −1. Finally, to correct for the sign of the DPF, the parties multiply their output with shares of $w$, the sign of the DPF provided by party $P_2$. This computation can be performed efficiently using one Beaver triple [7] provided by party $P_2$.

**Correctness.** The correctness of the protocol is easy to follow from the correctness of the DPF:

$$\begin{aligned} u_\sigma[i] &= y_\sigma[i+x] \\ &\neq 0 \;\text{ iff }\; i+x=r \end{aligned} \tag{4}$$

In other words, the last condition can be rephrased as iff $i=r-x=a$ (all equalities are modulo $2^k$), thus the PIR results in “selecting” the entry corresponding to $d_a$, as desired. Finally, the sign bit $w$, corrects the final answer for whether $y_0[r]-y_1[r]=+1$ or −1.

**Complexity.** Altogether, the party $P_2$ provides shares of $r$, the sign of the DPF and one Beaver triple, in addition to the DPF keys. In the semi-honest case, the shares of $r$ can simply be sent as one element of $\mathbb{Z}_{2^k}$ per party. The sign of the DPF $w$ is sent as one element of $\mathbb{Z}_{2^\ell}$ per party. The Beaver triple can be optimized by generating the random values using PRGs and thus only sending one element of $\mathbb{Z}_{2^\ell}$ per party[3]. Finally, with the tree-trimming optimization [42], the DPF keys themselves are $\lambda+(\lambda+2)\cdot\nu+2^{k-\nu}$ (where $\nu$ is defined by Eq. 34). Thus, the total communication is $2\ell+k+(\nu+2)\lambda+2\nu$ bits (note that $2^{k-\nu}\leqslant\lambda$).

The round complexity of our protocol is 3 rounds in total. The first round is the sharing of the correlated randomness and can be done in the offline/pre-processing phase. The second round is used to reconstruct the masked value $x$. And the final round is used for the beaver multiplication. Thus the *online round complexity is just 2 rounds.* Comparing this in terms of concrete efficiency, prior art requires 9-60 rounds [54, 71, 76, 88]. Furthermore, our communication protocol reduces communication by anywhere between 25× to two orders of magnitude.

# 5 Protocol for Malicious Security

In this section, we extend the semi-honest protocol ($\Pi_{\mathsf{sh:Func}}$) to be secure against malicious corruptions ($\Pi_{\mathsf{mal:Func}}$) in the standard UC Model [20]. Our central result here is a new result for malicious sketching (protocol to verify correctness of DPF keys) over the ring $\mathbb{Z}_{2^\ell}$. We prove the security of our protocol in Appendix B.

## 5.1 Protocol Overview

To make protocol in Fig. 1 secure against malicious adversaries with an honest majority corruption, we need to provably defend against adversarial generation of the DPF keys and other correlated randomness (assuming $P_2$ is corrupt) as well as prevent one of parties $P_0,P_1$ acting maliciously (assuming one of them is corrupt). To this end, we first describe how we modify $\Pi_{\mathsf{sh:Func}}$ to $\Pi_{\mathsf{mal:Func}}$ and then provide a proof for it's security (and correctness). The protocol is formally described in Fig. 2. We begin by describing the parameters used in the protocol followed by a brief description of secret sharing scheme.

**Parameters.** Throughout our protocol, we will use the following parameters: $\ell$, the size of the inputs (in bits) over which we wish to perform computation would typically be set to either 32 or 64. Similar to the semi-honest set-up, we consider that the underlying secrets are bounded within a range $(-2^{k-1},2^{k-1}]$ where practical values of $k\in[16,24]$. Finally, we use two different statistical security parameters $t$ and $s$. Given the security proofs and bounds offered by Lemma 5.5, 5.6, and 5.4, these parameters will be related to the desired level of statistical security $\mathsf{sec_s}$ as follows: $t=2s$, $s=\mathsf{sec_s}+1$ where $\mathsf{sec_s}$ is set to 40 or more.

**Notation.** Our starting point is the secret sharing scheme from the SPD$\mathbb{Z}_{2^k}$ protocol [29]. Each data item $x$ will consist of a secret sharing $[[x]]_{2^{\ell+t}}$ to denote secret sharing of $x$ which contains 3 components $(x_i,m_{xi},\alpha_i)$ where $x_i$ is the share, $m_{xi}$ is the MAC and $\alpha_i$ is the MAC key. The values $x_i,m_{xi}\in\mathbb{Z}_{2^{\ell+t}}$ and $\alpha_i\in\mathbb{Z}_{2^t}$ and the following relation holds for a valid sharing:

$$m_{x0}+m_{x1}\equiv(x_0+x_1)\cdot(\alpha_0+\alpha_1) \pmod{2^{\ell+t}} \tag{5}$$

Note that although $x\in\mathbb{Z}_{2^{\ell+t}}$, only the lower $\ell$-bits of $x$ matter and the MAC check protocols from [29] take this into account. For the broader MPC computation, we maintain the invariant that parties $P_0$,$P_1$ hold such a 2-out-of-2 secret sharing of each intermediate value. In other words, we consider the secret sharing scheme instantiated with the standard 2PC dishonest majority set-up. We show how our protocols can be modified to achieve malicious security using such a secret sharing scheme. Given two vectors $x$,$y$ of the same length, we use $\langle x,y\rangle$ to denote the inner product. We use $\equiv_m$ to denote congruence relations modulo $2^m$ and $\not\equiv_m$ for relations that do not hold. Thus, the congruence $a\equiv b \pmod{2^m}$ is abbreviated as $a\equiv_m b$. We use $[n]$ to denote the set $\{1,2,...,n\}$ and $S$ to denote the set $\{0,1,...,2^s-1\}$. Notation specific to the proofs will be mentioned within the proofs themselves.

[2] The database can also be shifted but for efficiency reasons, it is much easier to shift the vector $y$

[3] Note that all these communications can be further reduced by sending shares to a single party say $P_0$ and using PRGs on the other party $P_1$.

**Protocol Intuition.** The protocol follows the same high level idea as the semi-honest protocol described (Fig. 1). To analyze security against malicious clients, we break down the analysis into two cases: one of $P_0$,$P_1$ is malicious or $P_2$ is malicious. Below, we describe the challenges in each and how we overcome these challenges.

(1) *$P_0$ or $P_1$ is malicious.*This is the easier of the two cases. Here we know that the correlated randomness is well formed but the challenge is in ensuring that the parties correctly compute the function value. Our idea here is to use a dishonest majority secret sharing scheme to perform the computation between $P_0$,$P_1$. We then modify the DPF to provide keys that enable $P_0$,$P_1$ to locally expand not just shares of function value but also the shares of the MAC on the function values. We note that this can be achieved using a single DPF key with a larger payload, thus reducing the cost of the EvalAll (which is the bottleneck, cf. Section 6) by 2× compared to a naïve implementation.

(2) *$P_2$ is malicious.* In this case, the challenge lies in ensuring that the keys as well as the other correlated randomness provided by $P_2$ is "well formed." This is known as *Verifiable FSS* in the literature and corresponding verification algorithms are also known as malicious sketching schemes (based on the linear sketch used for the verification) [16]. However, all prior verifiable FSS schemes are over fields and the ones that operate over small rings such as $\mathbb{Z}_2$ simply cast the outputs into a larger field to perform the verification. However, since our protocols need to work over rings (including the MACs), we need a sketching scheme over rings. However, at the heart of the sketching scheme is the Schwartz-Zippel lemma [79, 92] which states that a random linear combination of ill-formed inputs will be caught with high probability. However, the SZ lemma does not hold over rings (refer to Section 1.1). To address this challenge, we state and prove a special form of SZ lemma over rings that enables us to design a maliciously secure sketching scheme suitable for our protocol.

Thus, the maliciously secure protocol $\Pi_{\mathsf{mal:Func}}$ follows the same skeleton as the semi-honest protocol $\Pi_{\mathsf{sh:Func}}$ but (1) changes the underlying secret sharing scheme (2) generates DPF keys with higher payload to also provide sharing of the MACs and (3) uses a maliciously secure sketching subroutine that $P_0$,$P_1$ run before the second part of the protocol. The entire protocol is described in Fig. 2. Next, we describe our sketching scheme in detail.

## 5.2 Malicious Sketching Scheme Over Rings

Our central contribution here is a technique for the parties $P_0, P_1$ to verify that the DPF keys provided by $P_2$ are well formed over rings, i.e., the keys on EvalAll expand into shares of two vectors $y$,$m_y$ such that both vectors are of hamming weight one (have a single non-zero entry) and $m_y = \alpha y$.

We use the same structure as the sketching protocol described in [8, 16] where a random linear combination is used to ensure that the output of EvalAll is one-hot. The expand on this to further use a procedure similar to the MAC Check protocol from the SPDZ-line of work [29, 34, 37] that enables detecting any malicious behaviour with high probability. We divide this into 2 components, a check to ensure correctness of the MACs and a check to ensure the correctness of the keys. However, the key ingredient for the latter, i.e., detecting ill-formed keys in prior sketching schemes is the Schwartz-Zippel lemma [79, 92], stated below informally:

**Lemma 5.1** (Schwartz-Zippel lemma informal)**.** *Given a non-zero multivariate polynomial of degree d, the probability that it evaluates to 0 over randomly chosen inputs is bounded by $d/|S|$ where S is any subset of the finite field over which the polynomial is defined.*

The lemma rests on the fact that a polynomial over a finite field cannot have more zeros than it's degree. However, such a statement does not hold over rings. For instance the polynomial $2^{\ell-1}x$ is a linear polynomial over the ring $\mathbb{Z}_{2^\ell}$ but has every even number as a zero. *Thus, the Schwartz-Zippel lemma in its current form does not hold over rings.* However, the SZ lemma is critical for the verification in the sketching schemes. In order to address this issue, we state and prove a new lemma that enables us to devise a sketching scheme over rings. Note that in order to ensure security and correctness against malicious behaviour, we require that $P_0$ and $P_1$ verify that the shares sent by party $P_2$ in $\Pi_{\mathsf{sh:Func}}$ are valid. Since we need the DPF keys to provide shares of the output as given by Eq. 5, we do not require the shares of $w$ to be shared by $P_2$. We achieve these goals in a one-shot computation and establish the following two informal statements:

(1) $k_\sigma$ form shares of a well formed DPF.
(2) $r_\sigma$ form shares of the index of the DPF.

Next we present the details of the sketching scheme.

## 5.3 Instantiating the Sketching Scheme

Let $N=2^k$ be the size of the DPF look-up. Suppose that the DPF keys shared by $P_2$ expand into the vectors $y,m_y$, each of size $N$. Now, the goals of the sketching schemes are twofold. First, given that the keys were generated honestly, the verification should succeed. And second, an invalid $y$ or $m_y$ should be accepted only with a negligible probability. We consider the following linear sketch $L \in \mathbb{Z}_{2^{\ell+t}}^{4\times N}$:

$$L=\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,N} \\ a_{2,1} & a_{2,2} & \dots & a_{2,N} \\ a_{3,1} & a_{3,2} & \dots & a_{3,N} \\ a_{4,1} & a_{4,2} & \dots & a_{4,N} \end{pmatrix} \tag{6}$$

where for $i\in[N]$, we have $a_{1,i},a_{2,i}$ randomly sampled from $\mathbb{Z}_{2^s}$ (in practice $P_0,P_1$ share PRG seeds and expand into these values), $a_{3,i} = a_{1,i}\cdot a_{2,i} \pmod{2^{\ell+t}}$ and $a_{4,i} = i-1$. For an intuitive understanding of the choice, refer to Eq. 8. Note that this matrix needs to be sampled only once and thus can be reused as long as no party aborts in the protocol. Let $L_1,L_2,L_3,L_4$ denote the rows of $L$.

**Verification.** Define $z_j = \langle L_j,y\rangle \in \mathbb{Z}_{2^{\ell+t}}$ for $j=1,2,3,4$. Furthermore, let $z^* = \langle L_1,m_y\rangle \in \mathbb{Z}_{2^{\ell+t}}$. Note that in practice, each party only holds a share of these values, i.e., $y$ (and thus each $z_j$) is secret shared over $\mathbb{Z}_{2^{\ell+t}}$ between $P_0,P_1$. The parties then evaluate the following expression (within MPC) and check if the value (as an element of $\mathbb{Z}_{2^{\ell+t}}$) is equal to 0, i.e.,

$$(z_1z_2-z_3)+(z_4-r)\overset{?}{=}0 \tag{7a}$$

$$(\alpha z_1 - z^*)\overset{?}{=}0 \tag{7b}$$

Parties $P_0,P_1$ run this verification over MPC and use this verification as an indication of correct/incorrect keys. We provide a more complete description of this MAC check over shares along with possible optimizations in Appendix E.

**Completeness.** The completeness of this sketching scheme is easy to verify. For honestly generated correlated randomness, $(z_4-r)$ by the construction of the DPF scheme equals zero (the DPF outputs 1 at $r$). At the same time, since $y$ is of hamming weight one, the first term is zero as shown below:

$$\begin{aligned} z_1z_2-z_3 &= \sum_i^N a_{1,i}a_{2,i}(y_i^2-y_i)+\sum_{i\neq j}^N a_{1,i}a_{2,j}(y_iy_j) \\ &=0 \end{aligned} \tag{8}$$

where the first term is zero because each $y_i \in \{0,1\}$ and the second term is zero because if at most one of $y_i,y_j$ is non-zero for any pair of distinct $i,j$. Thus each term equals zero for Eq. 7a. Finally, the verification Eq. 7b is valid by honest construction.

**Malicious protocol for function computation $\Pi_{\mathsf{mal:Func}}$**

**Auxiliary variables:** Parameters $\ell,k,f,t$ for bit-size, bound, fixed-point precision of inputs, and a statistical security parameter. The function $\mathsf{Func}$ and the function database $D_{\mathsf{Func}}$ of size $N=2^k$ with each entry $d_i$ in $\mathbb{Z}_{2^{\ell+t}}$ for $i\in[N]$.
**Inputs:** Secret shared values $[[a]]_{2^{\ell+t}}$.
**Outputs:** Secret shares $[[b]]_{2^{\ell+t}}$ where $b=\mathsf{Func}(a)$.

**Protocol:**

(0) Parties $P_0,P_2$ and $P_1,P_2$ invoke $\mathcal{F}_{\mathsf{Rand}}$ to generate values $r_0,r_1\in\mathbb{Z}_{2^{\ell+t}}$ and $\alpha_0,\alpha_1\in\mathbb{Z}_{2^t}$ such that $r_0,\alpha_0$ are known to $P_0,P_2$ and $r_1,\alpha_1$ are known to $P_1,P_2$. Party $P_2$ computes $r\equiv r_0+r_1 \pmod{2^{\ell+t}}$ and $\alpha\equiv\alpha_0+\alpha_1 \pmod{2^t}$. This step is non-interactive.

(1) Party $P_2$ generates the following correlated randomness:
  (a) Let $f_r$ be the a DPF with output 1 at $r$ and $g_r$ be the DPF with output $\alpha$ at $r$ (and 0 everywhere else). Use a single DPF with longer payload (over $\mathbb{Z}_{2^\ell}\times\mathbb{Z}_{2^{\ell+t}}$) to generate keys for the two DPFs $f_r$ and $g_r$, i.e.,

$$(k_0,k_1)\leftarrow\{\mathsf{Gen}(r,1),\mathsf{Gen}(r,\alpha)\}$$

  (b) Generate a sharing of $\alpha r$ over $\mathbb{Z}_{2^{\ell+t}}$. Send each share and the corresponding DPF key $k_\sigma$ to $P_\sigma$.

(2) Parties $P_0,P_1$ jointly sample a linear sketch $L\in\mathbb{G}^{4\times N}$ (cf Section 5.3 for details) where $\mathbb{G}=\mathbb{Z}_{2^{\ell+t}}$. Note that is generated one time and is private from $P_2$. For $\sigma=0,1$, party $P_\sigma$ computes the following within secure computation:
  (a) Set $[[r]]_{2^{\ell+t}}$ to be the output of $\mathcal{F}_{\mathsf{Rand}}$ along with the set of shares of $\alpha r$ sent by $P_2$.
  (b) Compute $y,m_y \leftarrow \mathsf{EvalAll}(\sigma,k_\sigma)$ where the vector $y=\{y[1],\dots,y[2^k]\}$ and $m_y=\{m_y[1],\dots,m_y[2^k]\}$
  (c) Set $z_j=\langle y,L_j\rangle$ for $j=1,2,3,4$ and $z^*=\langle m_y,L_1\rangle$.
  (d) Within the secure two party computation, the parties verify:

$$(z_1z_2-z_3)+(z_4-r)\in\mathbb{Z}_{2^{\ell+t}}$$

  equals 0 and if the MAC check succeeds on $(\alpha z_1-z^*)$. If it does, proceed to Step (3) below else $\mathsf{abort}$ the protocol.

(3) Party $P_\sigma$ for $\sigma\in\{0,1\}$ evaluates the following:
  (a) Perform an authenticated opening of $x\equiv r-a \pmod{2^k}$ using authenticated shares $[[r]]_{2^{\ell+t}}$ and $[[a]]_{2^{\ell+t}}$.
  (b) Set $u[i] = y[i+x]$ and $m_u[i] = m_y[i+x]$ for $i\in\{0,1,\dots,2^k-1\}$ and the addition is computed modulo $2^k$.
  (c) Compute $v=\langle u,D_{\mathsf{Func}}\rangle$, and $m_v=\langle m_u,D_{\mathsf{Func}}\rangle$ i.e.,

$$v=\sum_{i=0}^{2^k-1}u[i]\cdot d_i \quad\text{and}\quad m_v=\sum_{i=0}^{2^k-1}m_u[i]\cdot d_i$$

  where the product between an element of $\mathbb{Z}_{2^\ell}$ with an element of $\mathbb{Z}_{2^{\ell+t}}$ is done by static casting $\mathbb{Z}_{2^\ell}\to\mathbb{Z}_{2^{\ell+t}}$.
  (d) Invoke $\mathcal{F}_{\mathsf{Rand}}$ to generate two random values $\beta,m_\beta$ that are used to re-randomize $v,m_v$:

$$v\leftarrow v+(-1)^\sigma\beta \quad\text{and}\quad m_v\leftarrow m_v+(-1)^\sigma m_\beta$$

(4) Party $P_0,P_1$ output their share $c:=(v,m_v,\alpha_\sigma)$.

**Fig. 2.** Protocol for *maliciously secure* computation of a given function $\mathsf{Func}$ (such as division, exponentiation, tanh)

**Correctness.** The correctness of the protocol follows the exact same argument as that of the semi-honest version (for both the shares and the MACs).

**Soundness.** In order to establish the soundness of this verification scheme, we need to prove that for any invalid inputs $y, m^y$, the verification condition given by Eq. 7 is non-zero with high probability. We prove this in two steps, the first establishes the validity of $y$ and the second establishes the validity of $m_y$. The latter follows naturally from the standard MAC check routine with minimal modifications and the details are provided in Appendix E. To establish the former, we prove the following theorem:

**Theorem 5.2.** *Let $y \in \mathbb{Z}_{2^{\ell+t}}$ and let $t = 2s$. For a fixed value $r$, if either of these two statements is false:*
*(1) $y \pmod{2^\ell}$ is either a unit vector or the all-0 vector*
*(2) $\sum_{i\in[N]}(i-1)\cdot y_i \equiv_\ell r$*
*then, the probability that the secure computation check in Eq. 7a succeeds is bounded above by $2^{1-s}$.*

*Proof.* Note that when the computation is performed over a field $\mathbb{F}$, the Schwartz-Zippel lemma [79, 92] gives a failure probability $O(1/|\mathbb{F}|)$ which is small for sufficiently large field sizes. However, in our case, we need to prove a special form of the Schwartz-Zippel lemma that applies over rings. We first introduce the notion of a *s-distinct* set before we state our general lemma.

**Definition 5.3.** *Let $A$ be any subset of $\mathbb{Z}$ and let $s$ be a fixed positive integer. $A$ is said to be s-distinct if no two elements of $A$ are congruent modulo $2^s$. In other words, for any two values $x,y \in A$, $2^s \nmid x-y$.*

Note that the maximum cardinality of any *s-distinct* set is at most $2^s$ and we call such a set a *maximal s-distinct* set. The set $\{0,1,...,2^s-1\}$ is an example of a *maximal s-distinct* set. We state our generalization as the following lemma:

**Lemma 5.4.** ***(Bilinear Schwartz-Zippel lemma over Rings)*** *Let $p(x_1,...,x_n)$ be an $n$-variable polynomial over $\mathbb{Z}_{2^{\ell+2s}}$ of the following form:*

$$p(x_1,...,x_n) = \sum_{i\neq j} a_{ij}x_ix_j + \sum_i b_ix_i + c \tag{9}$$

*where the sums are over all indices in $[n]$ such that $p(\cdot)$ is not identically zero over $\mathbb{Z}_{2^\ell}$. Let $S$ be any $s$-*distinct *subset of $\mathbb{Z}_{2^{\ell+2s}}$. If points $y_1,...,y_n$ are sampled randomly and uniformly from $S$, then the probability that $p(y_1,...,y_n)$ evaluates to 0 over $\mathbb{Z}_{2^{\ell+2s}}$ is bounded above by $2^{1-s}$.*

We prove Lemma 5.4 in Section 5.4. Assuming Lemma 5.4, we can complete the proof of Theorem 5.2 by observing that the LHS of the condition given in Eq. 7a can be expanded out as:

$$\begin{aligned}\left(\sum_{i=1}^{N} a_{1,i}a_{2,i}(y_i^2-y_i)\right) + \left(\sum_{i\neq j} a_{1,i}a_{2,j}y_iy_j\right)\\ + \left(\sum_{i=1}^{N} a_{4,i}y_i\right) - r\end{aligned} \tag{10}$$

We observe that the above Eq. 10 is a bilinear function of the variables $\{a_{1,i},a_{2,i},a_{4,i}\}_{i=1}^N$. Furthermore, Step 0 (from Fig. 2) ensures that $r$ is chosen uniformly at random. Thus, assuming at least one of the conditions of Theorem 5.2 is true, the above bilinear polynomial is not identically zero over $\mathbb{Z}_{2^\ell}$ and using Lemma 5.4, we get the required bound on the success probability. □

## 5.4 Proof of Lemma 5.4

We split this proof into multiple parts. First we provide a high level proof structure, then describe and prove two intermediate lemmas, and finally use these lemmas together to prove the result.

**Proof Structure.** To establish Lemma 5.4, we prove two additional lemmas which informally can be stated as follows:

(1) A univariate version of the lemma that establishes the statement for an arbitrary polynomial degree. This is Lemma 5.5.
(2) A multi-variate version, that establishes the statement albeit only for linear polynomials. This is Lemma 5.6.

Finally, we combine these two lemmas to prove Lemma 5.4, the general $n$-variable, bilinear polynomial version. A more general quadratic form of the lemma is discussed in Appendix G.2.

**Univariate Lemma.** We first state and prove the lemma for a polynomial over a single variable for any degree $d$.

**Lemma 5.5.** *Let $p(x)$ be a polynomial of degree $d$ over $\mathbb{Z}_{2^{\ell+ds}}$ that is not identically zero over $\mathbb{Z}_{2^\ell}$. Then $p(x)$ has at most $d$ distinct roots in any $s$-*distinct *set, i.e., the set $R = \{x \mid p(x) = 0 \in \mathbb{Z}_{2^{\ell+ds}}\}$ has at most $d$ distinct values when reduced modulo $2^s$.*

*Proof.* We use $\equiv_m$ to denote congruence relations modulo $2^m$ and $\not\equiv_m$ for relations that do not hold (i.e., the congruence $a \equiv b \pmod{2^m}$ is abbreviated as $a \equiv_m b$). The statement holds trivially for the base case $d = 0$, i.e., constant polynomial in $\mathbb{Z}_{2^\ell}$ is either identically zero or has no roots. For the base case $d=1$, suppose that $p(x) = p_1x + p_0$. If $p_1 \equiv_\ell 0$ then, either $p_0 \equiv_\ell 0$ in which case we have the polynomial is identically zero in $\mathbb{Z}_{2^\ell}$, or $p_0 \not\equiv_\ell 0$, in which case it has no roots as $p_0 = p(r) - p_1 r \equiv_\ell 0$, a contradiction. If $p_1 \not\equiv_\ell 0$, then there exists at most one root in the set $S$. To see this, suppose $p_1 = 2^\nu p_{\mathsf{odd}}$ where $\nu < \ell$ and $p_{\mathsf{odd}}$ is odd. Then $r$ can be uniquely solved in $S$ as,

$$\begin{aligned} & p(r) \equiv_{\ell+s} 0 \\ \Rightarrow \quad & p_1 r \equiv_{\ell+s} -p_0 \\ \Rightarrow \quad & \quad r \equiv_{\ell+s-\nu} -p_0/p_{\mathsf{odd}} \\ \Rightarrow \quad & \quad r \equiv_s -p_0/p_{\mathsf{odd}} \end{aligned} \tag{11}$$

Below we provide a proof for $d=2$ that also provides the intuition for the general case (by sequential reduction) and defer the proof of the general case to Appendix A.

Suppose that $d=2$ and $p(x) = p_2 x^2 + p_1 x + p_0$ and that $p(x)$ is not identically 0 over $\mathbb{Z}_{2^\ell}$. We prove the lemma by contraposition. Suppose that $p(x)$ evaluates to 0 for points $r_1, r_2, r_3$ such that $r_i \in \mathbb{Z}_{2^s}$ for all $i \in [3]$ and $r_i \neq r_j$ for $i \neq j$. Then the system of equations $p(r_i) = 0$ can be written as:

$$\begin{pmatrix} r_1^2 & r_1 & 1 \\ r_2^2 & r_2 & 1 \\ r_3^2 & r_3 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_2 \\ p_1 \\ p_0 \end{pmatrix} \equiv_{\ell+2s} 0 \tag{12}$$

Linearly modifying the system of equations, we get

$$\begin{pmatrix} (r_1-r_3)(r_1+r_3) & (r_1-r_3) & 0 \\ (r_2-r_3)(r_2+r_3) & (r_2-r_3) & 0 \\ r_3^2 & r_3 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_2 \\ p_1 \\ p_0 \end{pmatrix} \equiv_{\ell+2s} 0 \tag{13}$$

Let $2^{\nu_1}, 2^{\nu_2}$ denote the highest powers of 2 dividing $(r_1 - r_3), (r_2 - r_3)$ respectively, i.e., $2^{\nu_i} \parallel (r_i - r_3)$ for $i = \{1,2\}$. Since $r_1, r_2, r_3$ are distinct in $\mathbb{Z}_{2^s}$, we know that $\nu_i < s$. Thus, if $\nu = \max\{\nu_1, \nu_2\}$, then the following system of equations hold:

$$\begin{pmatrix} r_1+r_3 & 1 & 0 \\ r_2+r_3 & 1 & 0 \\ r_3^2 & r_3 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_2 \\ p_1 \\ p_0 \end{pmatrix} \equiv_{\ell+2s-\nu} 0 \equiv_{\ell+s} 0 \tag{14}$$

Repeating the same process, we get that $p_2(r_1 - r_2) \equiv_{\ell+s} 0$ which implies that $p_2 \equiv_\ell 0$. Plugging this into the second constraint from Eq. 14, we get that $p_1 \equiv_\ell 0$ and finally combining these two statements with the third constraint in Eq. 14 we get that $p_0 \equiv_\ell 0$. Thus $p$ is identically zero over $\mathbb{Z}_{2^\ell}$ contradicting our initial assumption. This completes the proof for $d=2$. □

**Multivariate Lemma for Linear Polynomials.** Next we state and prove the lemma for linear polynomials over multiple variables.

**Lemma 5.6.** *Let $p(x_1,\dots,x_n)$ be a $n$-variable linear polynomial over $\mathbb{Z}_{2^{\ell+s}}$ such that it is not identically zero over $\mathbb{Z}_{2^\ell}$. Let $S$ be any maximal $s$-*distinct *set. If $y_1,\dots,y_n$ are sampled randomly and uniformly from $S$, then the probability that $p(y_1,\dots,y_n)$ evaluates to 0 over $\mathbb{Z}_{2^{\ell+s}}$ is bounded above by $2^{-s}$.*

*Proof.* Let $p(x_1,\dots,x_n) = c + \sum_i a_i x_i$. If $a_i \equiv_\ell 0$ for all $i \in [n]$, then either $p$ is identically zero in $\mathbb{Z}_{2^\ell}$ (when $c \equiv_\ell 0$) or $p$ has no roots (when $c \not\equiv_\ell 0$). Either way, the statement of the lemma is true. Thus, w.l.o.g let us assume that $a_1 \not\equiv_\ell 0$. Suppose that $\nu$ is the highest power of 2 dividing $a_1$, i.e., $2^\nu \parallel a_1$. Suppose that $y_2,\dots,y_n$ are randomly sampled values of $x_2,\dots,x_n$ from $S$. Let $y$ denote the following quantity:

$$y = -\left( c + \sum_{j=2}^{n} a_j y_j \right) \tag{15}$$

Then, $p(x_1, y_2, \dots, y_n) \equiv_{\ell+s} 0$ implies that:

$$\begin{aligned} & a_1 x_1 \equiv_{\ell+s} y \\ \Rightarrow \quad & \left(\frac{a_1}{2^\nu}\right) x_1 \equiv_{\ell+s-\nu} \frac{y}{2^\nu} \\ \Rightarrow \quad & \quad x_1 \equiv_{\ell+s-\nu} \left(\frac{a_1}{2^\nu}\right)^{-1} \cdot \frac{y}{2^\nu} \end{aligned} \tag{16}$$

where the inverse of $a_1/2^\nu$ exists because it is co-prime to the modulus. Now since $a_1 \not\equiv_\ell 0$, we know that $\nu < \ell$ and thus Eq. 16 shows that $x_1$ can be uniquely solved over *$s$-distinct* set $S$. Thus, for any random choice of $y_2,\dots,y_n$ there exist a single value $y_1$ of $x_1$ (among $|S|$ possibilities) that makes $p(y_1, \dots, y_n)$ evaluate to 0 making the overall probability $2^{-s}$ if $S$ is a *maximal $s$-distinct* set. This completes the proof of the lemma. □

**Proving Lemma 5.4 using Lemma 5.5, 5.6.** Finally, we establish Lemma 5.4 using the now established Lemma 5.5 and Lemma 5.6.

*Proof.* If $n = 1$, then this is simply a linear equation and thus has at most one zero in $\mathbb{Z}_{2^s}$ by Lemma 5.5. Suppose $n \geqslant 2$. Furthermore, suppose that $a_{ij} \equiv_{\ell+s} 0$ for all $i \neq j$. Then, $p(x_1, x_2, \dots, x_n) \equiv_{\ell+2s} 0$ can be reduced to the constraint $\tilde{p}(x_1, x_2, \dots, x_n) \equiv_{\ell+s} 0$ where $\tilde{p}(x_1, x_2, \dots, x_n) = \sum_i b_i x_i + c$. Using Lemma 5.6, we know that either $\tilde{p}$ is identically zero over $\mathbb{Z}_{2^\ell}$ (and consequently $p$) or the probability that $\tilde{p}(x_1, x_2, \dots, x_n) = 0$ for randomly sampled $x_1, x_2, \dots, x_n$ from $S$ is bounded above by $2^{-s} \leqslant 2^{1-s}$.

Thus, w.l.o.g, let us assume that $a_{12} \not\equiv_{\ell+s} 0$. We can re-write $p(x_1, x_2, \dots, x_n)$ as follows:

$$p(x_1,\dots,x_n) = x_1 q_1(x_2,\dots,x_n) + q_0(x_2,\dots,x_n) \tag{17}$$

where we know that $q_1(x_2,\dots,x_n)$ is a linear function of its variables and is not identically zero over $\mathbb{Z}_{2^\ell}$. Now let us randomly sample $y_2,\dots,y_n$ for $x_2,\dots,x_n$ from $S$ (note that $q_0$ is a constant for a given sample $y_2,\dots,y_n$). Since $q_1(\cdot)$ is not identically zero over $\mathbb{Z}_{2^\ell}$, Lemma 5.6 implies that

$$\Pr[q_1(y_2,\dots,y_n) \equiv_{\ell+s} 0] \leqslant \frac{1}{2^s} \tag{18}$$

On the contrary, if $q_1(y_2,\dots,y_n) \neq 0$ (i.e., $\not\equiv_{\ell+s} 0$), then the polynomial $\hat{p}(x_1) = p(x_1, y_2, \dots, y_n)$ is a univariate linear polynomial (not identically zero over $\mathbb{Z}_{2^\ell}$) and by Lemma 5.5

has at most one root. Thus,

$$\Pr[\hat{p}(x_1)=0 \mid q_1(y_2,\dots,y_n)\neq 0] \leqslant \frac{1}{2^{-s}} \tag{19}$$

If event $\mathcal{E}_1$ is $p(x_1, \dots, x_n) \equiv_{\ell+2s} 0$ and event $\mathcal{E}_2$ is $q_1(y_2,\dots,y_n) \equiv_{\ell+s} 0$ (where all variables are sampled uniformly from $S$) then,

$$\begin{aligned}\Pr[\mathcal{E}_1] &= \Pr[\mathcal{E}_1 \wedge \mathcal{E}_2] + \Pr[\mathcal{E}_1 \wedge \neg\mathcal{E}_2] \\ &= \Pr[\mathcal{E}_1 \wedge \mathcal{E}_2] + \Pr[\mathcal{E}_1 \mid \neg\mathcal{E}_2] \cdot \Pr[\neg\mathcal{E}_2] \\ &\leqslant \Pr[\mathcal{E}_2] + \Pr[\mathcal{E}_1 \mid \neg\mathcal{E}_2] \\ &\leqslant 2^{1-s}\end{aligned} \tag{20}$$

This completes the proof. □

# 6 Evaluation

In this section, we demonstrate how the theoretical contributions translate into concrete efficiency gains. For implementing the FSS, we use the libPSI/PIR library [74] and the Falcon codebase [88] is used for end-to-end testing and the neural network experiments. All our experiments are run over Azure Machines with the following configuration - Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz, 36 cores (72 threads), 64 GB of RAM. Our networking set-up includes LAN with a bandwidth of 10 Gbps and ping time of 0.2 ms and WAN with a bandwidth of 100 Mbps and 70 ms ping time. We use 16 threads for our experiments.

For comparison with prior work, we use the state-of-the-art protocols and implementations from MP-SPDZ [54] and ABY$^3$ [63]. In Appendix G.3, we also compare and show improvements against two other state-of-the-art protocols that use different adversarial models – SiRnn [71] and Falcon [88]. We compare the protocols in the semi-honest and malicious adversarial models and in the LAN and WAN settings. We also run experiments specific to neural networks to establish the importance of our protocols in scaling secure machine learning. All numbers reported for prior work were run on the same Azure set-up for an apples-to-apples comparison. Finally, we describe the optimizations used in Appendix G.1.

## 6.1 Comparison in the Semi-honest Setting

Our protocol is oblivious to the function in consideration and thus takes the same amount of time to compute normalization, sigmoid, tanh, logarithm, etc. We use the sigmoid function to enable comparison with the most number of prior state of the art frameworks. For the semi-honest setting, Table 1 presents a comparison of the proposed protocols against prior state-of-the-art works with varying batch sizes for $k=16$ and $l=32$ (effect of $k$ is studied in Appendix F). For MP-SPDZ (`replicated-ring-party.x`) we set the same 16 bit precision for `sfix`. To achieve similar precision, we use 19 piece approximation [78] for the ABY$^3$ comparison.

| | | Mode | $10^0$ | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ |
|---|---|---|---|---|---|---|---|---|
| Time | MP-SPDZ | LAN | 0.018 | 0.018 | 0.110 | 0.935 | 9.575 | 93.199 |
| | ABY$^3$ | | 0.090 | 0.092 | 0.096 | 0.104 | 0.221 | 1.445 |
| | Pika | | 0.0007 | 0.001 | 0.006 | 0.059 | 0.539 | 5.329 |
| | MP-SPDZ | WAN | 3.225 | 3.327 | 16.903 | 160.293 | 1605.170 | - |
| | ABY$^3$ | | 1.487 | 1.489 | 1.493 | 1.501 | 1.619 | 2.851 |
| | Pika | | 0.183 | 0.195 | 0.198 | 0.356 | 1.375 | 9.030 |
| Comm. | MP-SPDZ | | 0.163 | 0.183 | 1.271 | 12.705 | 126.493 | 1265.160 |
| | ABY$^3$ | | 0.046 | 0.124 | 0.945 | 9.234 | 92.327 | 923.252 |
| | Pika | | 0.0002 | 0.002 | 0.017 | 0.174 | 1.745 | 17.452 |

**Table 1.** Time (in seconds) and communication (in MB) for *semi-honest secure* protocols. The computation is a sigmoid function on a batch of size $10^0,10^1,\cdots,10^5$ in the LAN and WAN settings. MP-SPDZ ran over an hour for the largest batch size and the communication numbers reported are for the LAN setting.

As can be seen, our protocols generally outperform prior work in both the run-time and communication overhead. The amortized cost of computing *one single function evaluation is only* $53\mu s$ *and requires less than 190 Bytes* in the LAN setting. This is about 5× faster than ABY$^3$ and uses 23× less communication. Compared to the implementation in MP-SPDZ, this is over 119× faster and uses 153× less communication. In the WAN setting, our protocols outperform MP-SPDZ by one to three orders of magnitude and these improvements persist even for large batch sizes. ABY$^3$ which uses garbled circuits achieves comparable performance for large batch sizes. Finally, our protocols use 54×-74× lower bandwidth compared to prior state-of-the-art protocols.

Since our protocols are computation heavy, their performance shines for smaller batch sizes. In the LAN setting, this breakeven point occurs for batch sizes of about $10^2-10^3$; however in the WAN setting, our protocol outperforms prior work for batch sizes up to $10^4-10^5$. Note that typical sizes over which such function computation is performed in a neural network is small. For instance, Sigmoid is computed over vectors of size equal to the number of classes (10 for MNIST, CIFAR-10 datasets) and normalizations occur per batch or channel (between 32 to 512) – which is precisely the domain in which our protocols perform best.

## 6.2 Comparison in the Malicious Setting

Table 2 presents a comparison of our protocols with prior art for the malicious setting ($\ell=128, s=40$)[4]. In the LAN setting, when comparing against MP-SPDZ (`malicious-`

[4] The malicious protocol of ABY$^3$ was not implemented in their repo.

| | | Mode | $10^0$ | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ |
|---|---|---|---|---|---|---|---|---|
| Time | MP-SPDZ | LAN | 0.069 | 0.070 | 0.391 | 3.717 | 36.853 | 368.547 |
| | Pika | | 0.003 | 0.018 | 0.168 | 1.670 | 17.033 | 170.865 |
| | MP-SPDZ | WAN | 4.988 | 4.993 | 27.682 | 250.803 | 2513.140 | - |
| | Pika | | 0.203 | 0.218 | 0.370 | 2.299 | 25.186 | 196.726 |
| Comm. | MP-SPDZ | | 0.905 | 0.945 | 5.697 | 54.777 | 544.556 | 5445.560 |
| | Pika | | 0.0004 | 0.004 | 0.045 | 0.448 | 4.483 | 44.835 |

**Table 2.** Time (in seconds) and communication (in MB) for *maliciously secure* protocols. The computation is a sigmoid function on a batch of size $10^0$,$10^1$,···,$10^5$ in the LAN and WAN settings. ABY$^3$ [63] does not implement their malicious protocol.

rep-ring-party.x), the prior state-of-the-art implementation, our protocol is about 20× faster for a single function evaluation and the improvement reduces to 2× for larger batch sizes. However, when considering the WAN setting, our protocol is about 25× faster for smaller batch sizes and about two orders of magnitude faster for larger batch sizes.

This improvement can be attributed in part to the 2-3 orders of magnitude lower communication used by our protocol. Bulk of the overhead of the maliciously secure scheme is caused by (1) EvalAll not benefiting from the tree-truncation optimization (as the output is no longer a single bit) and (2) the large inner products to compute $z_1$,···,$z_4$,$z*$. These factors together make our protocol compute dominated and consequently, the protocol provides significant improvements in the WAN setting.

## 6.3 Machine Learning Applications

In this study, we show how our normalization can improve the scalability of machine learning algorithms. MPC protocols when used with fixed-point representation, provide inherently approximate computation when compared with plaintext computation over floating point data types. Without protocols such as normalization or sigmoid, the error accumulates with the depth of the computation and quickly blows up to cause integer overflows that render the computation meaningless. We show that adding normalization not only improves the convergence and final accuracy (known from ML) but crucially keeps the overall error in the network from snowballing across the depth of the network.

To establish this, we consider the simple LeNet architecture and modify it to increase the depth from the initial 4-layer network all the way up to an 8-layer network (a more detailed summary of the architectures is provided in Appendix C). We then initialize two networks: a plaintext network (computing over *64-bit floating point* data-types) and a corresponding secure computation network (computing over *32-bit fixed-point* data-types), trained over the MNIST dataset. We then compare the relative error, the accuracy, and the prediction confidence as a function of the depth. These results are shown with the green lines in Fig 3. As can be seen, the average relative error at each neuron output can grow to a staggering 400% with just an 8-layer network. At the same time, the accuracy and confidence also goes down significantly.

We then consider the same two networks, modified by adding normalization layers after the convolution layers as is typically done in machine learning. The impact of this modification is immediately obvious. These results are shown with orange lines in Fig 3. The average relative error remains nearly a constant independent of the depth. Furthermore, there is minimal/no loss in accuracy and confidence over deeper networks. Finally, the use of batch normalization also improves the training time, the final model accuracy, and the convergence (seen in the accuracies of Table 4).

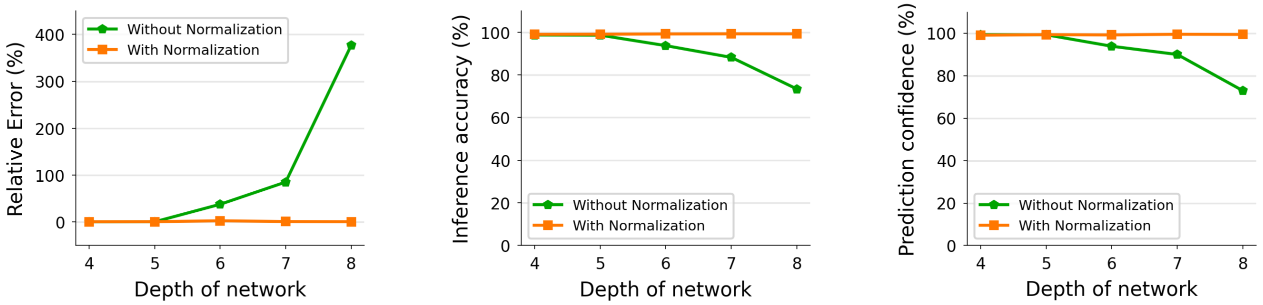
**Evaluation of DNNs.** We also evaluate our work on deep neural network architectures and compare against prior work. We consider two DNNs studied in the literature before – a Long short-term memory (LSTM) architecture from [61] which consists of 800 instances of sigmoid and 200 instances of tanh to predict next likely words and a benchmark from [75] which consists of two tanh layers with 2000 and 500 instances to classify sensor data into physical activities. As can be seen from the results in Table 3, our protocol improves the run-time by 8.13× using 53× less communication for the LSTM architecture. Over the DeepSecure DNN architecture, our work improves the runtime by over 35× while using merely 0.02% communication from the best prior work.

| Benchmark | Protocol | Runtime | Communication |
|---|---|---|---|
| MiniONN LSTM [61] | MiniONN | 1.1 | 182 MB |
| | SiRnn | 0.48 | 9.32 MB |
| | Pika | 0.059 | 0.1745 MB |
| DeepSecure B4 [75] | DeepSecure | 465 | 83.7 GB |
| | SiRnn | 5.3 | 1.94 GB |
| | Pika | 0.1489 | 0.4375 MB |

**Table 3.** Comparison with prior art over deep neural networks.

## 6.4 Measuring Computational Accuracy

ULPs or Units in Last Place is a way to measure the accuracy of approximate computation when using finite-bit representations such as floating-point or fixed-point [48]. It measures the maximum number of "representable numbers" between the rounding of the exact answer and the representation. This measure has a number of important robustness properties and hence is widely used in measuring the performance of standard math libraries such as GNU Scientific Library and Intel's libm [71]. Our protocols guarantee a maximum ULP error of 1 as the look-up table provides exact function computation within the given



**(a)** Normalization maintains computation accuracy over deeper networks.

**(b)** Normalization enables achieving high accuracy over deeper networks.

**(c)** Normalization ensures high prediction confidence over deep networks.

**Fig. 3.** These figures establish the importance of normalization for applications of secure computation. Without the use of normalization, the relative error quickly increases up with network depth (Fig 3a), the inference accuracy plummets (Fig. 3b), and so does the prediction confidence (Fig. 3c).

fixed-point precision – ABY$^3$ achieves 32 ULPs [63] of precision while MP-SDPZ that achieves 266 ULPs [38] for the same computation and SiRnn [71] and Falcon [88] achieve 3 and 42 ULPs respectively.

# 7 Related Work

Privacy-preserving protocols for non-linear functions such as normalization, exponentiation, and more widely secure computation over fixed point values was presented in these two seminal works [21, 22]. They gave a general framework of securely computing these protocols using approximate methods. A number of general purpose frameworks [1, 54] use these primitives (or the same protocol structure [30, 31, 57]) to implement these functions within secure computation. Below, we briefly describe some relevant prior works.

**Applications of Function Secret Sharing Schemes.** A recent closely related work [83] constructs a similar look-up based PIR protocol improved using FSS. However, their LUTs are populated with polynomial approximations in the vicinity of the inputs providing them with a construction that can tune the approximation errors to small values. Concurrent works SiRnn [71], AriaNN [76] share the goals of this work and focus on efficient primitives for non-linear operations used in machine learning. While the latter focuses on the use of FSS primarily for comparison, the former provides a new suite of protocols for an entire library of functions. Boyle *et al.* [9] also proposes the use of FSS for mixed-mode computation to improve the performance of ML functionalities. Another recent work by Boyle *et al.* [17] demonstrates the use of FSS to enable highly efficient protocols for secure comparisons. The work by Doerner and shelat [42] uses FSS to construct concretely efficient protocols for distributed ORAMs. Works such as [10–14] build upon the foundational work of Boyle *et al.* [16] to provide efficient pseudorandom correlation generators for commonly required correlated randomness such as Vector OLE or OT Extensions.

**3-Party Computation Frameworks.** A number of frameworks have focused on the 3-party computation models. The initial works in this space have been [63] which extends the original work on Arithmetic, Boolean, Yao interconversions [41] into a 3 party setting. The more recent ABY2.0 [68] provides improvements in the 2PC setting. SecureNN [86], Falcon [88] demonstrate efficient protocols for comparison using arithmetic secret sharing only. Other frameworks such as SWIFT, Astra, BLAZE [23, 58, 69] further show impressive theoretical and concrete efficiency gains. 4-party frameworks such as SWIFT, FLASH, Trident [19, 58, 70] provide stronger MPC guarantees such as fairness and guaranteed output delivery.

**Secure Computation over Rings.** The simplicity of implementations and consequently the concrete efficiency of protocols has made secure computation over finite rings an attractive design point in a number of recent frameworks. In addition to the works already mentioned above, there are a number of other works in this space. Work such as [29, 44] operate in the arithmetic blackbox model while other works [63, 64, 84, 85] focus on specific adversarial models and number of parties. Overdrive2k [67] provides efficient preprocessing in the dishonest majority setting which is relevant in tying this work with the broader MPC computation. A recent work by Vadapalli *et al.* [82] also provides an alternative approach to verification of FSS – known as an audit protocol. The idea there is to use secure computation to verify every level of the tree has exactly one identical child node and then expand the other node. However, their audit protocol requires the servers to be semi-honest whereas in Pika, one of the servers may be maliciously corrupt.

**Other MPC Frameworks.** In recent years, a number of frameworks have been proposed that focus on secure computation for machine learning applications. Works such as [59, 72] propose new protocols in 2PC, 3PC computation models tailored for neural networks along with an automated compiler for converting TensorFlow code into MPC code. The two recent works [44, 62] propose extremely efficient comparison protocols that are secure in the arithmetic blackbox model and thus hold over arbitrary corruptions. For dishonest majority setting, which is similar in spirit to our maliciously secure protocol, has a long line of work, most important and closely related are [5, 24, 29, 34, 37, 56, 67]. We refer the reader to this survey by Emmanuela Orsini [66] for more details. Other works such as Weng *et al.* [90], Yang *et al.* [91], and Baum *et al.* [6], show alternative ways to perform efficient secure computation that can be combined with standard iterative techniques to achieve concretely efficient protocols. This work can also benefit applications of MPC in other cryptographic applications [87].

## Acknowledgment

The authors would like to thank Aniket Kharkar for his invaluable feedback on the paper. I would also to like to thank all the anonymous reviewers who ever reviewed this work and special thanks to Dan Roche for his help with shepherding. Finally, I would like to thank the RISE Lab for the financial support.

## References

[1] Abdelrahaman Aly, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. SCALE-MAMBA v1.2: Documentation. https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf, 2018.

[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[3] V. Arvind, Partha Mukhopadhyay, and Srikanth Srinivasan. New Results on Noncommutative and Commutative Polynomial Identity Testing. In *Annual IEEE Conference on Computational Complexity*, 2008.

[4] Gilad Asharov, Shai Halevi, Yehuda Lindell, and Tal Rabin. Privacy-preserving search of similar patients in genomic data. In *Privacy Enhancing Technologies Symposium (PETS)*, 2018.

[5] Carsten Baum, Daniele Cozzo, and Nigel P Smart. Using topgear in overdrive: A more efficient zkpok for spdz. In *International Conference on Selected Areas in Cryptography*, 2019.

[6] Carsten Baum, Alex J. Malozemoff, Marc Rosen, and Peter Scholl. Mac'n'Cheese: Zero-Knowledge Proofs for Arithmetic Circuits with Nested Disjunctions. Cryptology ePrint Archive, Report 2020/1410, 2020. https://eprint.iacr.org/2020/1410.

[7] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[8] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[9] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *Advances in Cryptology—EUROCRYPT*, 2021.

[10] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector ole. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[11] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round ot extension and silent non-interactive secure computation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 291–308, 2019.

[12] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In *Advances in Cryptology—CRYPTO*, pages 489–518, 2019.

[13] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Correlated pseudorandom functions from variable-density lpn. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2020.

[14] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-lpn. In *Advances in Cryptology—CRYPTO*, 2020.

[15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Advances in Cryptology—EUROCRYPT*, 2015.

[16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[17] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In *Theory of Cryptography Conference (TCC)*, 2019.

[18] Paul Bunn and Rafail Ostrovsky. Secure two-party k-means clustering. In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[19] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. FLASH: Fast and robust framework for privacy-preserving machine learning. In *Privacy Enhancing Technologies Symposium (PETS)*, 2020.

[20] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2001.

[21] Octavian Catrina and Sebastiaan De Hoogh. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks*, 2010.

[22] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In *International Conference on Financial Cryptography and Data Security*, pages 35–50, 2010.

[23] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. Astra: High throughput 3pc over rings with application to secure prediction. In *ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019.

[24] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragoș Rotaru, Yongsoo Song, and Sameer Wagh. Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning. In *Advances in Cryptology—ASIACRYPT*, 2020.

[25] Shuo Chen, Jung Hee Cheon, Dongwoo Kim, and Daejun Park. Verifiable computing for approximate computa-

tion. Cryptology ePrint Archive, Report 2019/762, 2019. https://eprint.iacr.org/2019/762.

[26] Jung Hee Cheon, Dongwoo Kim, and Keewoo Lee. Mhz2k: Mpc from he over $Z_{2^k}$ with new packing, simpler reshare, and better zkp. Cryptology ePrint Archive, Report 2021/1383, 2021. https://ia.cr/2021/1383.

[27] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 1998.

[28] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[29] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPDZ2k: Efficient MPC mod $2^k$ for Dishonest Majority. In *Advances in Cryptology—CRYPTO*, 2018.

[30] Crypten: Privacy-preserving machine learning built on pytorch. https://github.com/facebookresearch/CrypTen, 2019.

[31] Morten Dahl, Jason Mancuso, Yann Dupis, Ben Decoste, Morgan Giraud, Ian Livingstone, Justin Patriquin, and Gavin Uhma. Private machine learning in tensorflow using secure computation, 2018.

[32] Anders Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security Symposium (USENIX)*, 2021.

[33] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[34] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical Covertly Secure MPC for Dishonest Majority–or: Breaking the SPDZ Limits. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.

[35] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology—CRYPTO*, pages 247–264. Springer, 2003.

[36] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The tinytable protocol for 2-party secure computation, or: Gate-scrambling revisited. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology—CRYPTO*, 2017.

[37] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[38] Data61. MP-SPDZ: Versatile Framework for Multi-party Computation. https://github.com/data61/MP-SPDZ, 2019.

[39] Ben DeCoste. Announcing securenn in tf-encrypted. https://medium.com/dropoutlabs/announcing-securenn-in-tf-encrypted-9c9c3e8a5a52, 2018.

[40] Richard A. Demillo and Richard J. Lipton. A probabilistic remark on algebraic program testing. *Information Processing Letters*, 7(4), 1978.

[41] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[42] Jack Doerner and abhi shelat. Scaling oram for secure computation. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[43] Z. Erkin, T. Veugen, T. Toft, and R. L. Lagendijk. Privacy-preserving user clustering in a social network. In *IEEE International Workshop on Information Forensics and Security (WIFS)*, 2009.

[44] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *Advances in Cryptology—CRYPTO*, 2020.

[45] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *USENIX Security Symposium (USENIX)*, 2021.

[46] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Advances in Cryptology—EUROCRYPT*, 2017.

[47] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Advances in Cryptology—EUROCRYPT*, 2014.

[48] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 1991.

[49] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *ACM Symposium on Theory of Computing (STOC)*, 1987.

[50] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3), 1996.

[51] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International Workshop on Artificial Neural Networks: From Natural to Artificial Neural Computation*, 1995.

[52] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, pages 448–456, 2015.

[53] Jonathan Katz, Samuel Ranellucci, Mike Rosulek, and Xiao Wang. Optimizing authenticated garbling for faster secure two-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology—CRYPTO*, 2018.

[54] Marcel Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[55] Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. Faster secure multi-party computation of aes and des using lookup tables. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *Applied Cryptography and Network Security (ACNS)*, 2017.

[56] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making SPDZ great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 158–189. Springer, 2018.

[57] B. Knott, S. Venkataraman, A.Y. Hannun, S. Sengupta, M. Ibrahim, and L.J.P. van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*, 2020.

[58] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. Swift: Super-fast and robust privacy-preserving machine learning. In *USENIX Security Symposium (USENIX)*, 2021.

[59] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[60] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[61] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious Neural Network Predictions via MiniONN Transformations. In *ACM Conference on Computer and Communications Security (CCS)*, pages 619–631, 2017.

[62] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient Comparison for Secure Multi-Party Computation. In *Financial Cryptography and Data Security (FC)*, 2021.

[63] Payman Mohassel and Peter Rindal. ABY$^3$: A mixed protocol framework for machine learning. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[64] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[65] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *ACM Conference on Electronic Commerce*, 1999.

[66] Emmanuela Orsini. Efficient, actively secure mpc with a dishonest majority: a survey. In *International Workshop on the Arithmetic of Finite Fields (WAIFI)*, 2020.

[67] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. Overdrive2k: Efficient secure mpc over $z_{2^k}$ from somewhat homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, 2019. https://eprint.iacr.org/2019/153.

[68] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security Symposium (USENIX)*, 2021.

[69] Arpita Patra and Ajith Suresh. Blaze: Blazing fast privacy-preserving machine learning. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[70] Rahul Rachuri and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[71] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. Sirnn: A math library for secure rnn inference. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[72] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[73] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2018.

[74] Peter Rindal. libPSI: A repository for private set intersection. https://github.com/osu-crypto/libPSI.

[75] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. DeepSecure: Scalable provably-secure deep learning. In *Annual Design Automation Conference*, 2018.

[76] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. Ariann: Low-interaction privacy-preserving deep learning via function secret sharing. In *Privacy Enhancing Technologies Symposium (PETS)*, 2022.

[77] Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. Secure multiparty pagerank algorithm for collaborative fraud detection. Cryptology ePrint Archive, Report 2018/917, 2018. https://eprint.iacr.org/2018/917.

[78] Jason Schlessman. Approximation of the sigmoid function and its derivative using a minimax approach, 2002. Master's thesis.

[79] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, 1980.

[80] Liyan Shen, Xiaojun Chen, Jinqiao Shi, Ye Dong, and Binxing Fang. An efficient 3-party framework for privacy-preserving neural network inference. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve Schneider, editors, *European Symposium on Research in Computer Security (ESORICS)*, 2020.

[81] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. Cryptgpu: Fast privacy-preserving machine learning on the GPU. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[82] A. Vadapalli, K. Storrier, and R. Henry. Sabre: Sender-anonymous messaging with fast audits. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[83] Adithya Vadapalli, Fattaneh Bayatbabolghani, and Ryan Henry. You may also like... privacy: Recommendation systems meet pir. In *Privacy Enhancing Technologies Symposium (PETS)*, 2021.

[84] Sameer Wagh. *New Directions in Efficient Privacy Preserving Machine Learning*. PhD thesis, Princeton University, 2020.

[85] Sameer Wagh. BarnOwl: Secure Comparisons using Silent Pseudorandom Correlation Generators. In *Tech Report*, 2022.

[86] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-Party Secure Computation for Neural Network Training. In *Privacy Enhancing Technologies Symposium (PETS)*, 2019.

[87] Sameer Wagh, Xi He, Ashwin Machanavajjhala, and Prateek Mittal. DP-Cryptography: marrying differential privacy and cryptography in emerging applications. In *Communications of the ACM*, 2020.

[88] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. In *Privacy Enhancing Technologies Symposium (PETS)*, 2021.

[89] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A GPU Platform for Secure Computation. In *USENIX Security Symposium (USENIX)*, 2022.

[90] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[91] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast Extension for coRRElated oT with small communication. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1607–1626, 2020.

[92] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, pages 216–226, 1979.

# A Generalized Proof of Lemma 5.5

*Proof.* We follow a similar line of argument as sketched in Section 5.4 for $d=2$. We prove the statement by contraposition. Suppose that

$$p(x) = \sum_{i=0}^{d} p_i x^i \in \mathbb{Z}_{2^{\ell+ds}}$$

such that not all $p_i \equiv_\ell 0$. Furthermore, suppose that $p(\cdot)$ evaluates to 0 at $d+1$ distinct points in $\mathbb{Z}_{2^s}$, i.e., there exists values $r_1,r_2,...r_{d+1}$ such that $p(r_i)=0$ and $r_i \in \mathbb{Z}_{2^s}$ for all $i \in [d+1]$ and $r_i \not\equiv_s r_j$ for $i \neq j$. Then the system of equations $p(r_i)=0$ can we written as:

$$\begin{pmatrix} r_1^d & r_1^{d-1} & \cdots & 1 \\ r_2^d & r_2^{d-1} & \cdots & 1 \\ \vdots & \vdots & & \vdots \\ r_{d+1}^d & r_{d+1}^{d-1} & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} p_d \\ \vdots \\ p_0 \end{pmatrix} \equiv_{\ell+ds} 0 \tag{21}$$

Let us denote the above Vandermonde matrix by $\mathcal{A}^{(0)}$ and the column vector as $P$. Then the above equation can be succinctly written as $\mathcal{A}^{(0)} \cdot P \equiv_{\ell+ds} 0$. The central idea of the proof is to perform a set of linear equations to simplify the above equations and to arrive at $p_i \equiv_\ell 0$ for all $i \in \{0,...,d\}$, contradicting our assumption. Before we describe the procedure in general, we demonstrate one step of the process and then formally describe the entire process. In the first step, we subtract the last row from every other row to reduce the set of equations to the following:

$$\begin{pmatrix} r_1^d - r_{d+1}^d & r_1^{d-1} - r_{d+1}^{d-1} & \cdots & 0 \\ r_2^d - r_{d+1}^d & r_2^{d-1} - r_{d+1}^{d-1} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ r_{d+1}^d & r_{d+1}^{d-1} & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} p_d \\ \vdots \\ p_0 \end{pmatrix} \equiv_{\ell+ds} 0 \tag{22}$$

Now note that for each $i \in [d]$, $r_i - r_{d+1}$ is a factor of all the elements of the row $i$. Thus, we can pull out the common factor. However, to cancel out this factor, we need to crucially observe that since this factor is not co-prime to the modulus $2^{\ell+ds}$, we have to reduce the modulus by their common factor. In particular, suppose $t_i = r_i - r_{d+1}$ for $i \in [d]$. Let $2^{\nu_i}$ denote the highest power of 2 dividing $t_i$, i.e., $2^{\nu_i} \| t_i$. Then we can divide row $i$ by $2^{\nu_i}$ and then the congruence relation holds modulo $2^{\ell+ds-\nu_i}$ where we divide $t_i/2^{\nu_i}$. Now since $t_i/2^{\nu_i}$ is odd and thus co-prime to $2^{\ell+ds-\nu_i}$, we know that it has an inverse modulo $2^{\ell+ds-\nu_i}$ and thus we can cancel it out. In summary, if $g(x,y,m)$ is defined as the following expansion:

$$g(x,y,m) = \frac{(x^m - y^m)}{(x-y)} = \sum_{i=0}^{m-1} x^i y^{m-1-i} \tag{23}$$

Let $f_{i,m} = g(r_i, r_{d+1}, m)$. We know because the values $r_i$ are distinct modulo $2^s$, if $\nu = \max\{\nu_1,...,\nu_d\}$ then $\nu < s$. Thus we can cancel a factor $\nu_i$ from row $i$ and still have the following constraints hold:

$$\begin{pmatrix} f_{1,d} & f_{1,d-1} & \cdots & 1 & 0 \\ f_{2,d} & f_{2,d-1} & \cdots & 1 & 0 \\ \vdots & \vdots & & & \vdots \\ f_{d,d} & f_{d,d-1} & \cdots & 1 & 0 \\ r_{d+1}^d & r_{d+1}^{d-1} & \cdots & r_{d+1} & 1 \end{pmatrix} \cdot \begin{pmatrix} p_d \\ \vdots \\ p_0 \end{pmatrix} \equiv_{\ell+(d-1)s} 0 \tag{24}$$

The goal of the proof is to establish that after repeating this process $d$ times, we arrive at the following form of constraints:

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ * & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & & \vdots & & \\ * & * & * & \cdots & 1 & 0 \\ * & * & * & \cdots & * & 1 \end{pmatrix} \cdot \begin{pmatrix} p_d \\ \vdots \\ p_0 \end{pmatrix} \equiv_\ell 0 \tag{25}$$

where * indicates terms we do not care about. Note that Eq. 25 immediately implies that $p_i \equiv_\ell 0$ for all $i \in \{0,1,...,d\}$ which contradicts our initial assumption that $p(\cdot)$ is not identically zero in $\mathbb{Z}_{2^\ell}$. Thus it remains to show that this sequential reduction arrives at Eq. 25. To do that we establish some further notation.

**Notation.** Let $\eta$ denote the steps, i.e., $\eta = 0$ refers to the constraint matrix $\mathcal{A}^{(0)}$ and given by the expression in Eq. 21 but $\eta = 1$ refers to the constraint matrix $\mathcal{A}^{(1)}$ given by the expression in Eq. 24. $\eta$ takes values from 1 to $d$ ($\eta = 0$ is defined for convenience). The process can be formally defined as following reduction:

**Procedure at iteration $\eta$**

For each $i \in \{1,2,...,d+1-\eta\}$ perform the following operations:

(1) Subtract $d+2-\eta^{\text{th}}$ row of matrix $\mathcal{A}^{(\eta-1)}$ from it's $i^{\text{th}}$ row.
(2) Pull out the common factor $r_i - r_{d+2-\eta}$ from the $i^{\text{th}}$ row.
(3) Cancel out the factor $r_i - r_{d+2-\eta}$ from the $i^{\text{th}}$ row, consequently reducing the modulus of the constraint system to $2^{\ell+(d-\eta)s}$

Thus, the matrix $\mathcal{A}^{(\eta)}$ at the end of step $\eta$ is a $(d+1) \times (d+1)$ matrix encoding the constraints, i.e.,

$$\begin{pmatrix} \mathcal{A}^{(\eta)}_{1,d} & \mathcal{A}^{(\eta)}_{1,d-1} & \cdots & \mathcal{A}^{(\eta)}_{1,0} \\ \mathcal{A}^{(\eta)}_{2,d} & \mathcal{A}^{(\eta)}_{2,d-1} & \cdots & \mathcal{A}^{(\eta)}_{2,0} \\ \vdots & \vdots & \vdots & \vdots \\ \mathcal{A}^{(\eta)}_{d+1,d} & \mathcal{A}^{(\eta)}_{d+1,d-1} & \cdots & \mathcal{A}^{(\eta)}_{d+1,0} \end{pmatrix} \cdot \begin{pmatrix} p_d \\ \vdots \\ p_0 \end{pmatrix} \equiv_{\ell+(d-\eta)s} 0 \tag{26}$$

We are interested in computing the $\mathcal{A}^{(\eta)}$ for $\eta = d$. Note that after step $\eta$, rows $1,2,...,(d+1-\eta)$ contain the same expression in their corresponding variables $r_i$, i.e., row $i$ can be transformed into row $j$ by replacing $r_i$ with $r_j$ for $i,j \in \{1,2,...,(d+1-\eta)\}$. Thus, to assist with computing these expressions, we define another $(d+1) \times (d+1)$ dimensional matrix $T^{(\eta)}$ to keep a track of each entry in the first row. In other words, $T^{(\eta)}$ has the following entries (note that the rows and columns of $T^{(\eta)}$ are differently numbers from

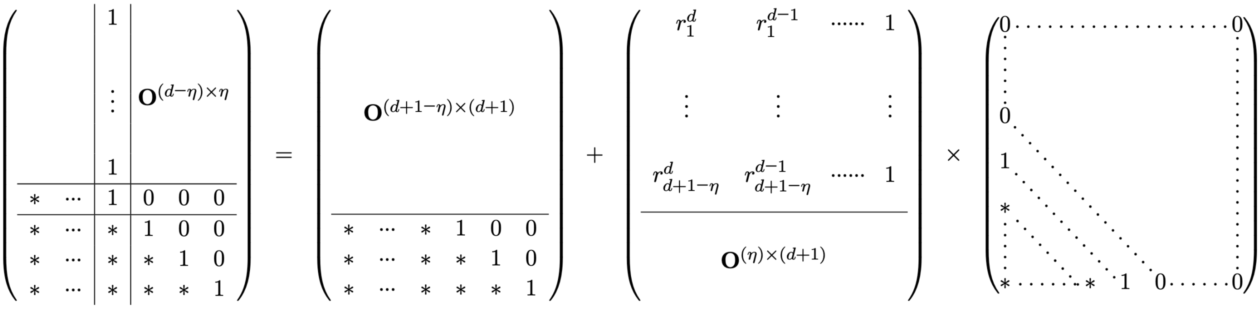


**Fig. 4.** To prove Claim A.2, we need to show that after the inductive step $\eta$, the matrix relation on the right hand side (the form of these is established because of the inductive hypothesis, Claims A.1, or by Eq. 33) imply the structure of the selected row and column on the left hand side as well as the zero matrix on the top right (zero matrices are denoted by $\mathbf{O}$)

the rows and columns of $\mathcal{A}^{(\eta)}$):

$$T^{(\eta)} = \begin{pmatrix} t^{(\eta)}_{d,1} & t^{(\eta)}_{d,2} & \cdots & t^{(\eta)}_{d,d+1} \\ t^{(\eta)}_{d-1,1} & t^{(\eta)}_{d-1,2} & \cdots & t^{(\eta)}_{d-1,d+1} \\ \vdots & \vdots & \vdots & \vdots \\ t^{(\eta)}_{0,1} & t^{(\eta)}_{0,2} & \cdots & t^{(\eta)}_{0,d+1} \end{pmatrix} \tag{27}$$

where, $t^{(\eta)}_{x,y}$ is the coefficient of the $y^{\text{th}}$ power of the root corresponding to that row (i.e., $r_i$) and corresponding to the entry in the $x^{\text{th}}$ column of $\mathcal{A}^{(\eta)}$. Note that the initial matrix $T^{(0)}$ is simply the identity matrix. For ease of exposition, we explicitly write down $T^{(1)}$ below:

$$T^{(1)} = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ r_{d+1} & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ r^{d-1}_{d+1} & r^{d-2}_{d+1} & r^{d-3}_{d+1} & \cdots & 0 \end{pmatrix} \tag{28}$$

Now, following the steps of the procedure at iteration $\eta$, we can compute coefficients corresponding to different powers of $r_i$ in $\mathcal{A}^{(\eta)}$ using a recurrence relation for $T^{(\eta)}$. In the $\eta^{\text{th}}$ step, we subtract the $(d+1-\eta)^{\text{th}}$ row of $\mathcal{A}^{(\eta-1)}$ from each row and then "divide" the $i^{\text{th}}$ row by $r_i - r_{d+1-\eta}$. We know that when we expand the expression for $g(r_i, r_{d+1-\eta}, m)$ for a general power $m$ (Eq. 23), each power of $r_i$ occurs exactly once. Thus, for computing the expression for $t^{(\eta)}_{x,y}$, we can combine the appropriate powers of $r_i$ and note that a contribution for power $x$ can only come from $t^{(\eta-1)}_{i,y}$ if $i > x$. Thus, we can write the following recurrence relation for $t^{(\eta)}_{x,y}$ for $\eta \in [d]$ (starting from the initial condition $T^{(0)} = \mathbb{I}$):

$$t^{(\eta)}_{x,y} = \sum_{i=x+1}^{d} t^{(\eta-1)}_{i,y} r^{i-x-1}_{d+2-\eta} \tag{29}$$

Calculating the general expression of $T^{(\eta)}$ is messy but we note that it is not necessary to compute that to establish $\mathcal{A}^{(d)}$ as given by the form in Eq. 25. We will use the following claims:

**Claim A.1.** *For $\eta \in [d]$, the coefficients of $T^{(\eta)}$ (given by Eq. 27) satisfies the following two conditions:*
*(1) $t^{(\eta)}_{x,y} = 0$ if $x+y \geqslant d+2-\eta$*
*(2) $t^{(\eta)}_{x,y} = 1$ if $x+y = d+1-\eta$*

To prove these claims, use use induction on $\eta$. First note that these conditions are satisfied for $T^{(0)}$ and $T^{(1)}$. Let us assume that these hold for $T^{(\eta-1)}$. Then using Eq. 29 if $x+y \geqslant d+2-\eta$ then for every $i \in \{x+1,...,d\}$

$$\begin{aligned} i+y &\geqslant x+1+y \\ &\geqslant d+3-\eta \\ &= d+2-(\eta-1) \end{aligned} \tag{30}$$

Thus every coefficient on the RHS of Eq. 29 is 0, thus establishing Claim A.1(1). Similarly, to prove Claim A.1(2), note that if $x+y = d+1-\eta$, then Eq. 29 can be split into the first summation term and the rest:

$$\begin{aligned} t^{(\eta)}_{x,y} &= t^{(\eta-1)}_{x+1,y} + \sum_{i=x+2}^{d} t^{(\eta-1)}_{i,y} r^{i-x-1}_{d+2-\eta} \\ &= t^{(\eta-1)}_{x+1,y} \end{aligned} \tag{31}$$

where the second equality follows from the fact that for $i \in \{x+2,...,d\}$, the coefficients are all zero $t^{(\eta-1)}_{i,y}$ as

$$\begin{aligned} i+y &\geqslant x+2+y \\ &= d+1-\eta+2 \\ &= d+2-(\eta-1) \end{aligned} \tag{32}$$

This establishes Claim A.1. Now, using the lower left triangular form of $T^{(\eta)}$ and noting that we only apply a transformation over rows in $[d+1-\eta]$, we can write the following recurrence relation for the entire matrix $\mathcal{A}^{\eta}$:

$$\mathcal{A}^{(\eta)} = \mathsf{Bottom}_{\eta}\left(\mathcal{A}^{(\eta-1)}\right) + \mathsf{Top}_{d+1-\eta}\left(\mathcal{A}^{(0)}\right) \times T^{(\eta)} \tag{33}$$

where $\mathsf{Bottom}_{\eta}(\cdot)$ retains only the bottom $\eta$ rows of the matrix and zeros the top $(d+1-\eta)$ rows and $\mathsf{Top}_{d+1-\eta}(\cdot)$ zeros the bottom $\eta$ rows and retains the top $(d+1-\eta)$ rows and $\mathcal{A}^{(0)}$ is simply the Vandermonde matrix in Eq. 21. The final set of claims to establish the form of $\mathcal{A}^{(d)}$ is the following:

**Claim A.2.** *For $\eta \in [d]$, the coefficients of $\mathcal{A}^{(\eta)}$ satisfy the following conditions:*
*(1) $\mathcal{A}^{(\eta)}_{i,j} = 1$ if $j \leqslant \eta$ and $i+j=d+1$*
*(2) $\mathcal{A}^{(\eta)}_{i,j} = 1$ if $j=\eta$ and $i+j \leqslant d+1$*
*(3) $\mathcal{A}^{(\eta)}_{i,j} = 0$ if $j<\eta$ and $i+j<d+1$*

Once again, we use induction to prove to the above claims. Please refer to Fig. 4 as an intuitive visual for proving these claims. The structure of the matrices is established either by the inductive hypothesis, Claims A.1, or by Eq. 33. First, we note that the bottom $\eta$ rows of $\mathcal{A}^{(\eta-1)}$ are unchanged thus establishing claim A.2(1) except for the case when $j=\eta$. We handle this case while establishing claim A.2(2).

For claim A.2(2), we note that the entries in this column are generated by multiplying the rows $\{1,2,...,d+1-\eta\}$ with the $(d+1-\eta)^{\text{th}}$ column of of $T^{(\eta)}$. Now for this column, using Claim A.1, we know that the only non-zero value is $t^{(\eta)}_{0,d+1-\eta}$ which is equal to 1 (note that the indices of $T^{(\eta)}$ are enumerated as shown in Eq. 27).

To establish claim A.2(3), we need to observe that for $y \in \{d+1-\eta,...,d+1\}$, the $y^{\text{th}}$ column of $T^{(\eta)}$ is all zero. This proves Claim A.2.

Finally, it is easy to see that Claim A.2 for $\eta = d$ implies the lower triangular matrix form of $\mathcal{A}^{(d)}$ with ones over the diagonal (as shown in Eq. 25), thus completing the proof of Lemma 5.5. □

## B End-to-end Security Proof

We prove security in the UC paradigm [20]. We first prove the security of the maliciously secure protocol and the semi-honest protocol follows from it. In the real interaction, parties $P_0,P_1,P_2$ run the protocol $\Pi_{\mathsf{mal:Func}}$ and in the ideal world, a simulator interacts with an ideal functionality (described below in Fig. 5) and the adversary (controlling 1-out-of-3 parties).

**Ideal functionality $\mathcal{F}_{\mathsf{Ideal}}$**

(1) Receive inputs $[[a]]_{2^{\ell+t}}$ from parties $P_0,P_1$.
(2) Wait to receive a signal from $\mathcal{S}$.
(3) If the $\mathcal{S}$ sends abort, send abort to the honest parties
(4) If the $\mathcal{S}$ sends OK, receive value $\alpha$ from $\mathcal{S}$, then compute $b=\mathsf{Func}(a)$, generate random secret shares of $b$, $\alpha b$ and send them to the honest parties.

**Fig. 5.** Description of the ideal functionality.

Let $\mathcal{F}_{\mathsf{Rand}}$ be a pairwise common randomness generation functionality. Note that such a functionality can be securely constructed using PRG keys set-up between parties [46]. The security can be formally stated as:

**Theorem B.1.** *Protocol $\Pi_{\mathsf{mal:Func}}$ securely computes the functionality $\mathcal{F}_{\mathsf{Ideal}}$ with abort in the $\mathcal{F}_{\mathsf{Rand}}$-hybrid model, in the presence of one malicious party.*

*Proof.* We break down the proof into two cases – (1) one of $P_0$ or $P_1$ is malicious or (2) $P_2$ is malicious.

**$P_0$ or $P_1$ is Malicious.** Without loss of generality, let us suppose the adversarially corrupt party is $P_0$ (denoted by $\mathcal{A}$). We construct a simulator $\mathcal{S}$ that interacts externally with the ideal functionality and internally with the adversary $\mathcal{A}$, as follows:

(1) $\mathcal{S}$ invokes $\mathcal{F}_{\mathsf{Rand}}$ to get random values for $\alpha_0,\alpha_1,r_0,r_1$.
(2) $\mathcal{S}$ generates the DPF keys honestly as described in Step (1)(a) and sends a share of $\alpha r$ and $k_0$ to $\mathcal{A}$.
(3) $\mathcal{S}$ also runs $\Pi_{\mathsf{mal:Func}}$ internally to simulate parties $P_1,P_2$. With the internal run, $\mathcal{S}$ proceeds to complete Step (2)(d) with $\mathcal{A}$. If the protocol aborts, $\mathcal{S}$ sends abort to the ideal functionality $\mathcal{F}_{\mathsf{Ideal}}$ and aborts.
(4) In the authenticated opening (Step (3)(a) in Fig. 2), the $\mathcal{S}$ uses $a_1 = 0$ (along with the random $r$ shares) as the share of the input and extracts the adversarial input $a_0$. It can then verify the MAC share using the extractor of the MAC check (and abort if incorrect MAC is used in the opening).
(5) If the protocol does not abort until this step, the $\mathcal{S}$ sends a OK signal (along with the value $\alpha$) to the ideal functionality and forwards the adversarial inputs to the ideal functionality. The output of the ideal functionality is forwarded as is to $\mathcal{A}$.

First, we show that all the real interaction transcripts can be simulated. The DPF keys and shares of $\alpha r$ are honestly generated and hence follow the same distribution in both the real and ideal world interactions. The opening and the commitments in Step (2)(d) and Step (3)(a) are masked by a uniformly random values that are only known to the simulator (hidden from $\mathcal{A}$) making the transcript indistinguishable in the real and ideal world scenario. The output distribution is indistinguishable in both the scenarios as the outputs form randomized shares of $b=\mathsf{Func}(a)$.

**$P_2$ is Malicious.** In this case, the adversarially corrupt party is $P_2$ (denoted by $\mathcal{A}$). We construct a simulator $\mathcal{S}$ that interacts externally with the ideal functionality and internally with the adversary $\mathcal{A}$, as follows:

(1) $\mathcal{S}$ invokes $\mathcal{F}_{\mathsf{Rand}}$ to get random values for $\alpha_0,\alpha_1,r_0,r_1$.

(2) $\mathcal{S}$ receives shares of $\alpha r$ and the DPF keys from $\mathcal{A}$, mimicking the roles of parties $P_0,P_1$ via an internal run.
(3) If $\mathcal{A}$ does not send valid shares of $\alpha r$ or the expansion of the DPF keys does not agree with the output of $\mathcal{F}_{\mathsf{Rand}}$, then $\mathcal{S}$ externally sends a signal abort to $\mathcal{F}_{\mathsf{Ideal}}$. Otherwise, $\mathcal{S}$ sends a signal OK to $\mathcal{F}_{\mathsf{Ideal}}$.

Note that $\mathcal{A}$ does not receive any outputs from the protocol. Thus the transcript of the protocol for $\mathcal{A}$ is indistinguishable in the real and ideal worlds. Thus we only need to establish that the honest parties receive their outputs correctly. First we note that the honest parties abort the protocol when Steps (2)(d) or (3)(a) fail. Theorem 5.2 along with the MAC check required for Step (3)(a) ensure that the real world protocol aborts under the same conditions as the ideal world protocol. Finally, the outputs received by the honest parties are random shares of $b=\mathsf{Func}(a)$ by the correctness argument in Section 5.3. □

## C Network Architectures

We use modifications of the LeNet architecture [60] for our experiments. Depending on the depth of the network, we will refer to these network architectures as 4-layer, 5-layer, ···, 8-layer. The first layer of each network is a convolution layer, with a $3\times3$ filter, stride and padding set to 1 each. The number of output filters is set to 8. The second layer is another convolution layer with a $3\times3$ filter, stride 1 and padding 1, with 8 input channels and 16 output channels. Each of these convolution layers are followed by a pooling layer (maxpooling) with a $2\times2$ filter. The final two layers are linear (fully connected) layers. The first of these is a 784 to 128 and the second one is 128 to 10. The output of the network is this size 10 (one-hot encoding) of the label. The activation function on each of these layers is ReLU.

| Layer Type | Input Size | Parameters | Activations | Repeat |
|---|---|---|---|---|
| Convolution | $28\times28$ | Filter size $3\times3$, Stride 1,<br>Padding 1, $\mathrm{C_{in}}=1$,$\mathrm{C_{out}}=8$ | $2\times2$ Maxpool<br>and ReLU | $1\times$ |
| Convolution | $14\times14\times8$ | Filter size $3\times3$, Stride 1,<br>Padding 1, $\mathrm{C_{in}}=8$,$\mathrm{C_{out}}=16$ | $2\times2$ Maxpool<br>and ReLU | $1\times$ |
| Convolution | $7\times7\times16$ | Filter size $3\times3$, Stride 1,<br>Padding 1, $\mathrm{C_{in}}=16$,$\mathrm{C_{out}}=16$ | ReLU | $(n{-}4)\times$ |
| Fully connected | 784 | $784\times128$ | ReLU | $1\times$ |
| Fully connected | 128 | $128\times10$ | ReLU | $1\times$ |

**Fig. 6.** Network architectures used in this work.

For the $n$-layer architecture, where $n\in\{4,5,6,7,8\}$, sandwiched between the two convolution layers and the two linear layers are $n-4$ convolution layers. These layers all use a $3\times3$ filter with stride 1 and padding 1 but with no pooling layer (as that sub-samples the data too much). In the networks that use batch normalization, the batch normalization is applied to the output of each of the convolution layers (before the pooling and the activation). These architectures are succinctly represented in Fig. 6.

## D Construction of the Function Secret Sharing Scheme

We briefly describe the FSS construction from [16] for the distributed point functions in Figure 7 (with the tree-trimming optimization). The evaluation depth of the tree is given by:

$$\nu=\min\left(\lceil n-\log\frac{\lambda}{\log|\mathbb{G}|}\rceil,n\right) \tag{34}$$

Fig. 8 describes the $\mathsf{Convert}_{\mathbb{G}}$ function for the specifics of our use case.

## E Details of the Sketching Scheme

Here we flush out the details of the sketching scheme with the underlying MPC computation. Let $\mathbb{G}_1$ be the group $\mathbb{Z}_{2^\ell}$ and $\mathbb{G}_2$ be the group $\mathbb{Z}_{2^{\ell+t}}$. Note that the first output of the DPF keys belongs to $\mathbb{G}_1$. To use this in the computation between $P_0,P_1$, we simply consider $\mathbb{G}_1$ as a subgroup of $\mathbb{G}_2$. This is fine since only the lower $\ell$-bits of the shares encode the secret. The MACs, which are the second output of the DPF keys, are already elements of $\mathbb{G}_2$.

Thus, we construct a single GGM tree that outputs two keys such that each key expands can be used to evaluate the function at a point and receive an output in $\mathbb{G}_1\times\mathbb{G}_2$. This would require modifying the final correction word $CW^{(\nu+1)}$ to return appropriately a pair of group elements. Note that given the sizes of $\ell,t$ we use, $\nu=n$, the full depth of the tree. Also, note that for the correctness of the MAC checks, $P_2$ encodes the value $\alpha$ times the first output over $\mathbb{Z}_{2^{\ell+t}}$ (and not just $\mathbb{Z}_{2^\ell}$). In this section, we use a superscript within parenthesis to denote the share of a party.

**Verification of Eq. 7a.** Party $P_\sigma$ for $\sigma=0,1$ locally expands their DPF keys into two vectors (using the EvalAll protocol):

$$y^{(\sigma)},m_y^{(\sigma)}\leftarrow\mathsf{EvalAll}(\sigma,k_\sigma,\mathsf{Pub}) \tag{35}$$

where $y^{(\sigma)}\in\mathbb{G}_1^N$ and $m_y^{(\sigma)}\in\mathbb{G}_2^N$. Then, party it locally computes the following terms:

$$z_j^{(\sigma)}=\langle y^{(\sigma)},L_j\rangle \quad \text{for} \quad j=1,2,3,4 \tag{36}$$

```
FSS Protocol for DPF (Gen,EvalAll)

Let $G:\{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ be a pseudorandom generator.
Let $\mathsf{Convert}_{\mathbb{G}}$ as defined in Figure 8 be a map that converts a
random $\lambda$-bit string into $\lfloor \lambda/m \rfloor$ pseudorandom elements of $\mathbb{G}$. Let
$\nu = \min\left(\lceil n - \log\frac{\lambda}{\log|\mathbb{G}|}\rceil, n\right)$

Gen($\alpha,\beta$):
  Let $\alpha=\alpha_1,\ldots,\alpha_n \in \{0,1\}^n$ be the bit decomposition of $\alpha$
  Sample random $s_0^{(0)} \leftarrow \{0,1\}^\lambda$ and $s_1^{(0)} \leftarrow \{0,1\}^\lambda$
  Let $t_0^{(0)}=0$ and $t_1^{(0)}=1$
  for $i$ = 1 to $\nu$ do
      $s_b^L||t_b^L||s_b^R||t_b^R \leftarrow G(s_b^{(i-1)})$ for $b=0,1$
      if $\alpha_i=0$ then
          Keep $\leftarrow L$, Lose $\leftarrow R$
      else
          Keep $\leftarrow R$, Lose $\leftarrow L$
      end
      $s_{CW} \leftarrow s_0^{\mathsf{Lose}} \oplus s_1^{\mathsf{Lose}}$
      $t_{CW}^L \leftarrow t_0^L \oplus t_1^L \oplus \alpha_i \oplus 1$   and   $t_{CW}^R \leftarrow t_0^R \oplus t_1^R \oplus \alpha_i$
      $s_b^{(i)} \leftarrow s_b^{\mathsf{Keep}} \oplus t_b^{(i-1)} \cdot s_{CW}$ for $b=0,1$
      $t_b^{(i)} \leftarrow t_b^{\mathsf{Keep}} \oplus t_b^{(i-1)} \cdot t_{CW}^{\mathsf{Keep}}$ for $b=0,1$
      $CW^{(i)} = s_{CW}||t_{CW}^L||t_{CW}^R$
  end
  Let $\hat{\alpha}=\alpha_{\nu+1},\ldots,\alpha_n$
  Let $CW^{(\nu+1)} =$
    $(-1)^{t_1^{(\nu)}}\left[e_{\hat{\alpha},\beta}^{2^{n-\nu}} - \mathsf{Convert}_{\mathbb{G}}(s_0^{(\nu)}) + \mathsf{Convert}_{\mathbb{G}}(s_1^{(\nu)})\right]$
  Let $k_b \leftarrow s_b^{(0)}$ for $b=0,1$ and Pub $=CW^{(1)}||\ldots||CW^{(\nu+1)}$
  return $(k_0,k_1,\mathsf{Pub})$

EvalAll($b,k_b$,Pub):
  return Traverse($k_b,b$,Pub,$\nu+1,1$)

Traverse($s,t$,Pub,$i,j$):
  Parse Pub $=CW^{(1)}||\ldots||CW^{(n+1)}$
  if $i \geqslant 1$ then
      Parse $CW^{(i)} = s_{CW}||t_{CW}^L||t_{CW}^R$
      $\tau^{(i)} = G(s) \oplus t \cdot \left(s_{CW}||t_{CW}^L||s_{CW}||t_{CW}^R\right)$
      Parse $\tau^{(i)} = s^L||t^L||s^R||t^R$
      return Traverse($s^L,t^L$,Pub,$i-1,j$)
        || Traverse($s^R,t^R$,Pub,$i-1,j+2^{n-\nu+i-1}$)
  else
      $P_{j'} = \mathsf{Convert}_{\mathbb{G}}(s+t\cdot CW^{\nu+1})[j']$
        for $j'=j,\ldots,j+2^{n-\nu}+1$
      return $P_j||P_{j+1}||\ldots||P_{j+2^{n-\nu}+1}$
  end
```

**Fig. 7.** Function Secret Sharing scheme for DPFs

```
Subroutine Convert_G

Convert_G($s$):
  Let $m=\log|\mathbb{G}|$, $b=\lfloor \lambda/m \rfloor$ and parse $s=s_1,\cdots s_\lambda \in \{0,1\}^\lambda$
  Let $f$ be a map that converts $m$-bits into an element of $\mathbb{G}$
  return $f(s_1,\ldots,s_m),\cdots,f(s_{bm-m+1},\ldots,s_{bm})$.
```

**Fig. 8.** Converting a string $s \in \{0,1\}^\lambda$ to an element in a group $\mathbb{G}$. We only use $\mathsf{Convert}_{\mathbb{G}}$ in the scenario when $\log|\mathbb{G}| \leqslant \lambda$.

While this two-party computation can be performed entirely between $P_0,P_1$, we can use further correlated randomness provided by $P_2$ to aid this computation. Such an optimization is akin to using a Beaver triple [7] and is used in prior works [8, 17]. More specifically, $P_2$ provides the parties with correlated randomness $b_1^{(\sigma)},b_2^{(\sigma)},b_3^{(\sigma)},b_4^{(\sigma)},b_5^{(\sigma)}$ which the parties use to reveal the values $Z_j \leftarrow z_j+b_j$ for $j=1,2,3,4$ and $R \leftarrow r+b_5$. Finally, $P_2$ also supplied the parties with shares of $A,B,C$ given by:

$$\begin{aligned} A &= -b_2 \\ B &= -b_1 \\ C &= b_1b_2+b_3-b_4+b_5 \end{aligned} \tag{37}$$

Note that the checking equation then modifies to the following:

$$\begin{aligned} (z_1z_2-z_3)+(z_4-r) = Z_1Z_2-Z_3+Z_4-R \\ +Z_1[-b_2]+Z_2[-b_1]+[b_1b_2+b_3-b_4+b_5] \end{aligned} \tag{38}$$

Thus, the servers open the masked values and verify the above expression over two rounds. Note that any malicious errors added either by $P_2$ or one of $P_0,P_1$ will be caught in this check. Thus the only attack that the corrupt party can run is a denial of service, which we consider outside the scope of this work.

**Verification of Eq. 7b.** The MAC check equation cannot be verified in the same equation as the verification for Eq. 7a. This is because, Lemma 5.4 would only allow us to detect cheating if $m_y - \alpha y \not\equiv_\ell 0$. However, for the verifications of the SPD$\mathbb{Z}_{2^k}$ framework [29] to work with statistical failure probability corresponding to our parameters, ensuring $m_y - \alpha y \equiv_\ell 0$ is not sufficient. We can overcome this challenge by simply running the MAC check routine. However, the MAC checks from [29], viz., SingleCheck,BatchCheck are protocols from opening and thus require careful masking of higher order bits. In our case, we do not wish to reveal any value but only verify the check and thus we can simply use a single authenticated value $b,m_b \in \mathbb{Z}_{2^{\ell+t}}$ to verify this check. Thus, the parties reveal $Z_1 \leftarrow z_1+b$, commit and reveal the following values:

$$\begin{aligned} \alpha z_1 - z^* &= \alpha(Z_1-b)-z^* \\ &= \alpha(Z_1)-\alpha b-z^* \\ &= \sum_{\sigma\in\{0,1\}} Z_1\alpha^{(\sigma)} - m_b^{(\sigma)} - z^{*(\sigma)} \end{aligned} \tag{39}$$

The last expression can be computed locally by each party after the reveal. Note that we can optimize the reveals across the two verifications by using $b=b_1$. Finally, we can also verify MAC on $r$ within the same expression by appending $r$ (and $m_r$) to $y$ (and $m_y$ resp.) but can also be deferred to the following secure computation.

**Complexity.** The shares for $b_1,b_2,b_3,b_4,b_5$ can simply be generated using PRG seeds shared one time between $P_\sigma$ and $P_2$ for $\sigma=0,1$. Thus their amortized communication overhead can be ignored. Ignoring the optimization of sending shares only to one party (of $A,B,C$), the communication is 3 group elements (thus $3(\ell+t)$-bits).

The verification itself (both Eq. 7a, 7b) can be done over two rounds. The first involves opening the masked values and the second to perform the check over the computed values. The opening involves a communication of 5 group elements and the computed values are one group element for each verification. Thus, the total communication is 7 group elements (thus $7(\ell+t)$-bits) split over two rounds.

# F Microbenchmarks for Experimental Evaluation

Tables 4 contains the results of training neural networks with varying number of layers with and without normalization. Table 5 contains the breakdown of the cost of our protocols as a function of the FSS tree size and compute and communication. An important observation from Table 5 is that our protocols are heavily compute dominated. For protocols in Falcon [88], the overhead of the normalization layers increases 44× when going from LAN to WAN (for secure inference; 290× for training). On the contrary, the overhead of our protocol remains nearly constant across LAN or WAN (final columns of Table 5). Once again, this is because our protocols have extremely low communication overhead and round complexity and thus are heavily compute dominated. Thus, these protocols, being compute bound, can further benefit by dedicated hardware implementations such as over GPUs or FPGAs [81, 89].

# G Discussion

Here we discuss other aspects of our protocol such as the optimizations used in our implementation, discussion on the SZ Lemma and generalizations, and a comparison of our work with other state-of-the-art protocols.

## G.1 Optimizations

Since our protocols are compute heavy, we use two optimizations from literature to improve our performance, viz., tree-trimming and optimized EvalAll. The first is an algorithm known as EvalAll that evaluates the function at all the points in its domain. This optimized EvalAll reduces the computational complexity over a naïve approach of running Eval at each point by $O(\log N)$. We describe this with along with the FSS scheme from [16] in Appendix D.

The second optimization is known as *tree-trimming* and is useful when the size of the output group $\mathbb{G}$ is smaller than the $\lambda$, where $\lambda$ is the size of the length-doubling PRG used in the FSS constructions. In such a case, the tree-trimming technique allows extracting multiple group elements using a single seed and thus the tree is evaluated up to a depth that is smaller than the actual depth $n$ by $\lambda/\log|\mathbb{G}|$ levels. Since the bulk of the overhead occurs in the lowest few levels of the evaluation, this optimization considerably speeds up evaluation. Over and above this, we parallelize the evaluation of the FSS keys across a batch to optimize the run-time of our protocol. We use 8 threads in our benchmarks to parallelize this computation. Note that the inner product currently is implemented using a simple for loop. It can be further optimized for larger batches by implementation using efficient matrix multiplication algorithms.

## G.2 Schwartz-Zippel Lemma and Generalizations

The Schwartz-Zippel lemma, also known as the DeMillo-Lipton-Schwartz–Zippel was originally stated in [40, 79, 92] (it appears that only [79] had the stronger result which is the widely accepted statement of the lemma). Since then, there have been a few works that try to extend this to rings [3, 25]. However, each approach requires some relaxation to the original statement such as finding sets $A$ with the condition that $\forall x,y\in A, x\neq y \implies x-y$ is not a zero divisor of the modulus or finding large integer domains. For the particular choice of ring in this work, such sets are not large enough to provide meaningful statistical security and hence are not useful for our setting.

In Section 5, we stated and proved a special form (billinear) of the Schwartz-Zippel lemma which suffices to construct maliciously secure protocol for look-up style computation. However, it remains an open question to state and prove a general statement (for arbitrary degree polynomials) of the lemma (particularly without significantly higher slack), similar to the original Schwartz-Zippel lemma. For instance, we can prove a SZ lemma for degree $d=2$ and for a general $n$ with an additional slack as follows:

**Lemma G.1.** ***(Quadratic Schwartz-Zippel lemma over Rings)*** *Let $p(x_1,\dots,x_n)$ be a $n$-variable polynomial over $\mathbb{Z}_{2^{\ell+4s}}$ and such that $p(\cdot)$ is not identically zero over $\mathbb{Z}_{2^\ell}$. Let $S$ be a* maximal $s$-distinct *set. If points $y_1,\dots,y_n$ are sampled randomly and uniformly from $S$, then the*

| | | | 4-Layer | 5-Layer | 6-Layer | 7-Layer | 8-Layer |
|---|---|---|---|---|---|---|---|
| Without Normalization | | Relative error | **0.31** | **0.44** | **37.83** | **85.16** | **376.75** |
| | Training | Plaintext Acc. | 99.20 | 99.32 | 99.41 | 99.40 | 99.55 |
| | Inference | Plaintext Acc. | 98.72 | 98.64 | 98.41 | 98.37 | 98.39 |
| | | MPC Acc. | 98.72 | 98.64 | 93.72 | 88.21 | 73.39 |
| | | Difference | **0.00** | **0.00** | **-4.69** | **-10.16** | **-25.00** |
| | | Confidence | 99.38 | 99.32 | 93.91 | 90.03 | 72.95 |
| With Normalization | | Relative error | **0.61** | **0.79** | **2.59** | **1.09** | **0.73** |
| | Training | Plaintext Acc. | 99.60 | 99.65 | 99.78 | 99.87 | 99.92 |
| | Inference | Plaintext Acc. | 99.01 | 99.08 | 99.20 | 99.25 | 99.25 |
| | | MPC Acc. | 99.01 | 99.08 | 99.20 | 99.25 | 99.25 |
| | | Difference | **0.00** | **0.00** | **0.00** | **0.00** | **0.00** |
| | | Confidence | 99.13 | 99.37 | 99.24 | 99.55 | 99.46 |

**Table 4.** Detailed results of the experiments establishing the use of normalization. The numbers in bold establish the major observations: without the use of normalization, the relatively error quickly accumulates to high values with the increasing depth of the network. This can be kept nearly constant with the use of normalization. The inference accuracy matches exactly the accuracy of the underlying network if normalization techniques are used.

| Tree Size (k) | Comm. (KB) | Time (ms) | | | | | |
|---|---|---|---|---|---|---|---|
| | | EvalAll | Compute | Network | | Total | |
| | | | | LAN | WAN | LAN | WAN |
| 16 | 18.71 | 3.98 | 1.26 | $1.5\times10^{-5}$ | $4.6\times10^{-4}$ | 5.25 | 5.25 |
| 17 | 20.33 | 7.17 | 1.23 | $1.6\times10^{-5}$ | $5.0\times10^{-4}$ | 8.40 | 8.41 |
| 18 | 21.96 | 10.89 | 1.62 | $1.7\times10^{-5}$ | $5.4\times10^{-4}$ | 12.51 | 12.51 |
| 19 | 23.58 | 15.15 | 7.47 | $1.8\times10^{-5}$ | $5.9\times10^{-4}$ | 22.62 | 22.62 |
| 20 | 25.21 | 25.90 | 18.03 | $2.0\times10^{-5}$ | $6.3\times10^{-4}$ | 43.93 | 43.93 |
| 21 | 26.83 | 48.34 | 35.04 | $2.1\times10^{-5}$ | $6.7\times10^{-4}$ | 83.39 | 83.39 |
| 22 | 28.46 | 120.07 | 49.06 | $2.2\times10^{-5}$ | $7.1\times10^{-4}$ | 169.13 | 169.13 |
| 23 | 30.08 | 196.50 | 148.55 | $2.4\times10^{-5}$ | $7.5\times10^{-4}$ | 345.06 | 345.06 |
| 24 | 31.71 | 365.56 | 320.40 | $2.5\times10^{-5}$ | $7.9\times10^{-4}$ | 685.96 | 685.96 |

**Table 5.** Breakdown of the overhead of our protocol for any function look-up as a function of the parameter $k$. The timings are for a batch size of 100. EvalAll contains the time for DPF evaluations and forms the bottleneck, compute is the rest of the compute cost (primarily the inner product), and network is the communication overhead of the protocols. Observe that the protocols are purely compute dominated and show very little increase when implemented over network constraint settings such as WAN.

*probability that $p(y_1,\ldots,y_n)$ evaluates to 0 over $\mathbb{Z}_{2^{\ell+4s}}$ is bounded above by $2^{1-s}$.*

*Proof.* The proof follows a similar line of argument to the proof of Lemma 5.4. Suppose one of the quadratic coefficients is $\not\equiv_{\ell+2s} 0$. Then using Lemma 5.5 over that variable (with $d \to 2$ and $\ell \to \ell+2s$) gives a failure probability of $2^{1-s}$. On the other hand, if all the quadratic coefficients are $\equiv_{\ell+2s} 0$, then we can reduce the equation $p(\cdot) \equiv_{\ell+4s} 0$ to $\tilde{p}(\cdot) \equiv_{\ell+2s} 0$, where $\tilde{p}$ is billinear (because all the quadratic terms are zero over $\mathbb{Z}_{2^{\ell+2s}}$). Using Lemma 5.4, the probability is bounded by $2^{1-s}$. Thus in either case, the probability is bounded by $2^{1-s}$, thus establishing the result. □

The most useful features of the Schwartz-Zippel lemma for rings are (1) the independence of the bound on the number of variables $n$ and (2) the scaling of the slack with the degree $d$. For $d=2$, we can achieve the independence with $n$ with a slack of $4s$ but it is an open question if tighter bounds could exist. Thus it remains to see if there is such a "tight" formulation of the lemma for rings and for higher polynomial degrees (possibly using sets coprime to the modulus). Another promising future direction of research is the construction of malicious sketching schemes that run the sketching only over a smaller selected domain. This would significantly reduce the cost of the inner product and would require leveraging the structure of the FSS schemes to prove correctness with high probability.

## G.3 2PC and Other Adversarial Models

Two recent works, SiRnn and Falcon, also provide protocols for non-linear function computation but operate in slightly different adversarial models. SiRnn is a 2PC semi-honest secure protocol which uses digit decomposition in conjunction with oblivious transfers to compute non-linear functions efficiently. Falcon on the other hand operates in the same 3PC adversarial model as this work with an

| | | Mode | $10^0$ | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ |
|---|---|---|---|---|---|---|---|---|
| Time | SiRnn | LAN | - | 0.115 | 0.116 | 0.147 | 0.289 | 2.089 |
| | Falcon | | 0.024 | 0.021 | 0.022 | 0.041 | 0.159 | 1.228 |
| | Pika | | 0.0007 | 0.001 | 0.006 | 0.059 | 0.539 | 5.329 |
| | SiRnn | WAN | - | 7.683 | 7.690 | 7.816 | 9.313 | 32.702 |
| | Falcon | | 3.226 | 3.232 | 3.253 | 3.373 | 4.340 | 13.577 |
| | Pika | | 0.183 | 0.195 | 0.198 | 0.356 | 1.375 | 9.030 |
| Comm. | SiRnn | | - | 0.897 | 0.988 | 2.159 | 23.800 | 238.085 |
| | Falcon | | 0.0014 | 0.014 | 0.137 | 1.396 | 13.960 | 139.600 |
| | Pika | | 0.0002 | 0.002 | 0.017 | 0.174 | 1.745 | 17.452 |

**Table 6.** Broader comparison against state-of-the-art *semi-honest* secure protocols. Note that SiRnn is a 2PC protocol and Falcon is a 3PC protocol but reveals the size of the input leading to simpler protocols. Time (in seconds) and communication (in MB) and the computation is a sigmoid function on a batch of size $10^0$,$10^1$,···,$10^5$ in the LAN and WAN settings.

| | | Mode | $10^0$ | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ |
|---|---|---|---|---|---|---|---|---|
| Time | Falcon | LAN | 0.0453 | 0.0473 | 0.0477 | 0.0867 | 0.4673 | 4.0399 |
| | Pika | | 0.003 | 0.018 | 0.168 | 1.670 | 17.033 | 170.865 |
| | Falcon | WAN | 7.6574 | 7.6630 | 7.6686 | 7.8132 | 11.0553 | 36.8217 |
| | Pika | | 0.203 | 0.218 | 0.370 | 2.299 | 25.186 | 196.726 |
| Comm. | Falcon | | 0.0100 | 0.1004 | 1.0036 | 10.0360 | 100.3600 | 1003.6000 |
| | Pika | | 0.0004 | 0.004 | 0.045 | 0.448 | 4.483 | 44.835 |

**Table 7.** Broader comparison against state-of-the-art *maliciously* secure protocols. Note that Falcon is a 3PC protocol but reveals the size of the input leading to simpler protocols (SiRnn [71] does not implement their malicious protocol). Time (in seconds) and communication (in MB) and the computation is a sigmoid function on a batch of size $10^0$,$10^1$,···,$10^5$ in the LAN and WAN settings.

honest majority corruption model (both semi-honest and malicious corruptions). However, Falcon protocol achieve a different ideal functionality and the computation thus reveals the nearest power of 2. This is frequently the bottleneck of the non-linear function computation leading to different scaling of the protocol with larger batch sizes.

Table 6 shows how this work compares against SiRnn and Falcon in the semi-honest threat model. In the LAN setting, our work improves upon these for smaller batch sizes. In the WAN setting however, our protocols unconditionally improve upon prior work. Finally, our protocols use 7×-727× lower bandwidth compared to either of these protocols.

**Constructing a 2PC Protocol.** Protocols described in Section 4, 5 are described as 3 party protocols and have been proven secure in a honest majority setting with semi-honest and malicious security respectively. These protocols can be converted into the weaker adversarial model of 2PC with a trusted third party such as those used in [64, 73, 76].

To convert our protocol into a 2-party computation protocol, we can use prior work in the semi-honest setting. Particularly, distributed generation of the FSS keys can be done with concrete efficiency by the protocol of Doerner and shelat [42] (see also Appendix A in [9]). The additional common randomness of generating the shares of the DPF point $r$ can be done using the input to the distributed FSS key generation (which have boolean shares of the $r$) and use works such as [33, 41, 44] to convert into arithmetic shares of $r$. Another option is to use concretely-efficient general-purpose secure computation protocols such as [1, 53, 54] to emulate the party $P_2$ (these outperform [42] for larger domain sizes). We use the implementation by Doerner and shelat [42] to benchmark the timing required for a distributed FSS key generation. In the LAN setting, it takes $202\mu s$ and in the WAN it requires $50.26ms$ for a single generation. Implementation for larger batch sizes can be optimized to improve run-times. Currently, the distributed key generations are sequential (since they are implemented inside an ORAM read functionality) thus rendering the round complexity linear in the size of the batch. Despite the round sub-optimal implementation, the overall run-times of Pika are better/comparable to SiRnn as seen in Table 8.

| | | $10^0$ | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ |
|---|---|---|---|---|---|---|---|
| LAN | Pika-2PC | 0.0009 | 0.0019 | 0.0122 | 0.1197 | 1.1642 | 11.6794 |
| | SiRnn | - | 0.1147 | 0.1158 | 0.1471 | 0.2886 | 2.0887 |
| WAN | Pika-2PC | 0.2337 | 0.2482 | 0.2761 | 0.6721 | 4.0554 | 33.9804 |
| | SiRnn | - | 7.6830 | 7.6904 | 7.8164 | 9.3129 | 32.7022 |

**Table 8.** Run-time (in seconds) comparison between a 2PC semi-honest construction using Pika with the protocol from SiRnn.

The question of enabling the malicious protocol in the 2PC case is harder to answer. Firstly, since there is no notion of party $P_2$ being untrusted (it is simply between party $P_0, P_1$ that the FSS keys are generated), it may be possible to absorb the verification routine into the MPC. The second challenge is generating the double payload FSS keys where neither party has access to the MAC key $\alpha$. Finally, the question whether such a protocol provides concrete efficiency improvements over state of the art protocols remains to be seen.

## G.4 Quantization of Inputs

We have seen how the techniques presented in this work can enable efficient computation of non-linear functions such as sigmoid, normalization, logarithm, square root etc. The improvements are most pronounced when (1) the inputs are/can be transformed into a smaller range (2) the computation is typically required over smaller batches. Thus, while the second is application dependent (and we have seen holds well in typical machine learning applications), we explore techniques that can enable the inputs to be transformed into smaller domains (this is driven by practical numbers used in machine learning, refer to Section 2.1, 3 and 6).

In applications such as machine learning, good learning practices typically have try to ensure that output activations are transformed to follow a normal distribution $\mathcal{N}(0,1)$, i.e., a Gaussian distribution with 0 mean and variance 1. In this case, the inputs are roughly equal to the size of the floating precision and in such cases, our protocols are ideally suited and directly applicable. However, in case the bounds on the inputs is large, i.e., $|x| \leqslant 2^m$ where $k < m < \ell$, then we need additional techniques to use our protocols. In order to enable a larger range of values, we need to reduce the size of the fixed-point precision. This can be achieved by standard truncation techniques described below:

**Semi-honest Security.** In the semi-honest setting, we locally truncate the input $a$ using the result from [64] with one bit loss in accuracy. More specifically, we set $[a]_{2^k} \leftarrow \Pi_{\mathsf{sh:Trunc}}([a]_{2^\ell}, \sigma, m-k)$ and then reconstruct $y \equiv a+r \pmod{2^k}$ where $\Pi_{\mathsf{sh:Trunc}}$ denotes the following protocol:

$$\Pi_{\mathsf{sh:Trunc}}(x,\sigma,t) = \begin{cases} x \gg t & \text{if } \sigma = 0 \\ -(-x \gg t) & \text{if } \sigma = 1 \end{cases} \tag{40}$$

where $\gg$ denotes a bit-shift operation. Note that Theorem 1 from [64] establishes that if $x \in \mathbb{Z}_{2^\ell}$, $|x| \leqslant 2^m$ for $k \leqslant m \leqslant \ell - 2$, then $\Pi_{\mathsf{sh:Trunc}}(x,\sigma,m-k)$ for $\sigma = 0,1$ are shares of $x/2^{m-k}$ with at most 1 bit error with probability $1 - 2^{m+1-\ell}$. Setting the value $m+1-\ell$ small enough, this probability can be made small enough. Once the value is truncated, the fixed-point precision is set to 0 if $f \leqslant m-k$ and set to $f-m+k$ otherwise. Thus we can then use our protocols to compute the function on this "quantized" value.

**Malicious Security.** In the case of malicious security, we can use the truncation protocols to reduce the size of the fixed-point precision. To this end, we can use a maliciously secure truncation protocols $\Pi_{\mathsf{mal:Trunc}}$, which can be instantiated using a general maliciously secure protocol such as [44] (for probabilistic but more efficient protocols Fig. 9 and for deterministic truncation Fig. 10 from [44]). Alternatively, we can also use bit-decomposition protocols from from [33] or digit-decomposition from [71] to extract appropriate chunks of the input. Once these are completed, the rest of the process is similar to the semi-honest case with function computed with appropriate fixed-point precision.

## G.5 Protocols Over Finite Fields

While the central contribution of this work is to enable protocols over rings, the ideas can be directly used for constructing protocols over fields as well. The fields setting has natural advantages in the malicious setting when compared against the ring setting, viz., the elimination of the slack in the representation. Suppose that the field is $\mathbb{F}_{2^\lambda}$, then the size of the FSS payload can be reduced to simply the $\lambda+1$ where the single bit is used to extract the bit for the information retrieval and is considered as the additive subgroup of $\mathbb{F}_{2^\lambda}$ and the MAC is stored as an element of $\mathbb{F}_{2^\lambda}$. In contrast, the size of the FSS payload in the ring case is $\ell + t = 2\ell + 2\mathsf{sec_s} + 2$. Combining other slack elimination protocols such as [62], entire ML computations can now be performed over 64-bit data types whereas prior work using such techniques typically require a 128-bit data representation.